CHAPTER 11

Controlling the Number of Threads Used for Execution

By default, the TBB library initializes its scheduler with what is typically the right number of threads to use. It creates one worker thread fewer than the number of logical cores on the platform, leaving one of the cores available to execute the main application thread. Because the TBB library implements parallelism using tasks that are scheduled on to these threads, this is usually the right amount of threads to have – there is exactly one software thread for each logical core, and the scheduling algorithms in TBB efficiently distribute tasks to these software threads using work stealing as described in Chapter 9.

Nevertheless, there are many scenarios in which we may justifiably want to change the default. Perhaps we are running scaling experiments and want to see how well our application performs with different numbers of threads. Or perhaps we know that several applications will always execute on our system in parallel, so we want to use only a subset of the available resources in our application. Or perhaps we know that our application creates extra native threads for rendering, AI, or some other purpose and we want to restrict TBB so that it leaves room on the system for those other native threads. In any case, if we want to change the default, we can.

There are three classes that can be used to influence how many threads participate in executing a specific TBB algorithm or flow graph. The interactions between these classes can be very complicated though! In this chapter, we focus on the common cases and best-known practices that will likely be enough for all but the most complicated applications. This level of detail will be sufficient for most readers, and the recommendations we make will be enough for almost all situations. Even so, readers

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_11

who want to understand the lowest level nuts-and-bolts of TBB are welcome to wade into the weeds in the TBB documentation to get into all of the details of the possible interactions between these classes if they choose. But if you follow the patterns outlined in this chapter, we don't think that will be necessary.

A Brief Recap of the TBB Scheduler Architecture

Before we begin talking about controlling the number of threads used in executing parallel algorithms, let's refresh our memory on the structure of the TBB scheduler shown in Figure 11-1. A more in-depth description of the TBB scheduler is found in Chapter 9.

The global thread pool (market) is where all of the worker threads start before migrating to task arenas. Threads migrate to task arenas that have tasks available to execute, and if there are not enough threads to fill all of the slots in all of the arenas, the threads fill slots in proportion to the number of slots in the arenas. For example, a task arena with twice as many slots as another arena will receive roughly twice as many workers.

Note If task priorities are in use, worker threads will fully satisfy the requests from task arenas with higher priority tasks before filling slots in task arenas with lower priority tasks. We discuss task priorities in more detail in Chapter 14. For the rest of this chapter, we assume all tasks are of equal priority.

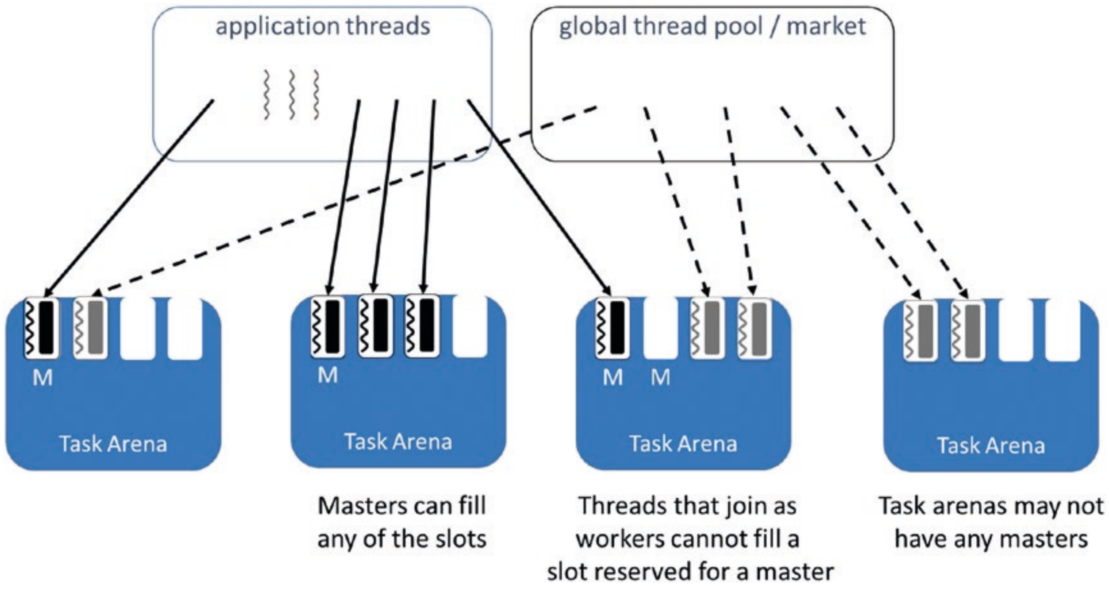

application threads
global thread pool / market
M
Task Arena
M
Task Arena
M M
Task Arena
Task Arena
Masters can fill any of the slots
Threads that join as workers cannot fill a slot reserved for a master
Task arenas may not have any masters

Figure 11-1. *The architecture of the TBB task scheduler*

Task arenas are created in one of two ways: (1) each master thread gets its own arena by default when it executes a TBB algorithm or spawns tasks and (2) we can explicitly create task arenas using `class task_arena` as described in more detail in Chapter 12.

If a task arena runs out of work, its worker threads return to the global thread pool to look for work to do in other arenas, or to sleep if there's no work in any arena.

Interfaces for Controlling the Number of Threads

The TBB library was first released over a decade ago, and it has evolved over that time to keep pace with the evolution of platforms and workloads. Now, TBB offers three ways to control threads: `task_scheduler_init`, `task_arena`, and `global_control`. In simple applications, we might be able to use just one of these interfaces to accomplish everything we need, but in more complex applications, we may need to use a combination of these interfaces.

Controlling Thread Count with `task_scheduler_init`

When the TBB library was first released, there was only a single interface for controlling the number of threads in an application: `class task_scheduler_init`. The interface of this class is shown in Figure 11-2.

A task_scheduler_init object can be used to (1) control when the task arena associated with a master thread is constructed and destroyed; (2) set the number of worker slots in that thread's arena; (3) set the stack size for each worker thread in the arena; and, if needed, (4) set an initial *soft* limit (see the side bar) on the number of threads available in the global thread pool.

```
class task_scheduler_init {
public:
  static const int automatic = /* implementation defined */;
  static const int deferred = /* implementation defined */;

  task_scheduler_init(int number_of_threads=automatic,
                      stack_size_type thread_stack_size=0);
  ~task_scheduler_init();

  void initialize(int number_of_threads=automatic);
  void initialize(int number_of_threads,
                  stack_size_type thread_stack_size);
  void terminate();

  static int default_num_threads ();
  bool is_active() const;

  // preview features
  void blocking_terminate();
  bool blocking_terminate(const std::nothrow_t&)
noexcept(true);
};
```

Figure 11-2. *The task_scheduler_init class interface*

Controlling Thread Count with task_arena

Later, as TBB was used on larger systems and in more complex applications, class task_arena was added to the library to create *explicit* task arenas as a way to isolate work. Work isolation is discussed in more detail in Chapter 12. In this chapter, we focus on how class task_arena lets us set the number of slots available in those explicit arenas. The functions in class task_arena used in this chapter are shown in Figure 11-3.

Using the `task_arena` constructor, we can set the total number of slots in the arena using the `max_concurrency` argument and the number of slots reserved exclusively for master threads using the `reserved_for_masters` argument. When we pass a functor to the `execute` method, the calling thread attaches to the arena, and any tasks spawned from within the functor are spawned into that arena.

```
class task_arena {
public:
  static const int automatic = implementation-defined;

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

  template<typename F> auto execute(F& f) -> decltype(f());
  template<typename F> auto execute(const F& f) ->
decltype(f());
  template<typename F> void enqueue(F&& f);
  template<typename F> void enqueue(F&& f, priority_t p);
};
```

Figure 11-3. *The `task_arena` class interface*

SOFT AND HARD LIMITS

The global thread pool has both a *soft limit* and a *hard limit*. The number of worker threads available for parallel execution is equal to the minimum of the soft limit value and the hard limit value.

The soft limit is a function of the requests made by the `task_scheduler_init` and `global_control` objects in the application. The hard limit is a function of the number of logical cores, `P`, on the system. At the time of the writing of this book, there is a hard limit of 256 threads for platforms where `P <= 64`, 4P for platforms where `64 < P <= 128`, and 2P for platforms where `P > 128`.

TBB tasks are executed non-preemptively on the TBB worker threads. So, oversubscribing a system with many more TBB threads than logical cores doesn't make a lot of sense – there are just more threads for the OS to manage. If we want more TBB threads than the *hard* limit allows, it is almost guaranteed that we are either using TBB incorrectly or trying to accomplish something that TBB was not designed for.

Controlling Thread Count with `global_control`

After `class task_arena` was introduced to the library, TBB users began requesting an interface to directly control the number of threads available in the global thread pool. The `class global_control` was only a *preview feature* until TBB 2019 Update 4 (it is now a full feature - meaning it is available by default without needing to enable with a preview macro definition) and is used to change the value of global parameters used by the TBB task scheduler – including the soft limit on the number of threads available in the global thread pool.

The class definition for `class global_control` is shown in Figure 11-4.

```
class global_control {
public:
  enum parameter {
    max_allowed_parallelism,
    thread_stack_size
  };
  global_control(parameter p, size_t value);
  ~global_control();
  static size_t active_value(parameter param);
};
```

Figure 11-4. *The `global_control` class interface*

Summary of Concepts and Classes

The concepts used in this chapter and the effects of the various classes are summarized in this section. Don't worry too much about understanding all of the details presented here. In the next section, we present best-known methods for using these classes to achieve specific goals. So, while the interactions described here may appear complicated, typical usage patterns are much simpler.

The scheduler: The TBB scheduler refers to the global thread pool and at least one task arena. Once a TBB scheduler is constructed, additional task arenas may be added to it, incrementing a reference count on the scheduler. As task arenas are destroyed, they decrement the reference count on the scheduler. If the last task arena is destroyed, the TBB scheduler is destroyed, including the global thread pool. Any future uses of TBB tasks will require construction of a new TBB scheduler. There is never more than one TBB scheduler active in a process.

The hard thread limit: There is a hard limit on the total number of worker threads that will be created by a TBB scheduler. This is a function of the hardware concurrency of the platform (see **Soft and Hard Limits** for more details).

The soft thread limit: There is a dynamic soft limit on the number of worker threads available to a TBB scheduler. A `global_control` object can be used to change the soft limit directly. Otherwise, the soft limit is initialized by the thread that creates the scheduler (see **Soft and Hard Limits** for more details).

The default soft thread limit: If a thread spawns a TBB task, whether directly by using the low-level interface or indirectly by using a TBB algorithm or flow graph, a TBB scheduler will be created if none exists at that time. If no `global_control` objects have set an explicit soft limit, the soft limit is initialized to `P-1`, where `P` is the platform's hardware concurrency.

`global_control` objects: A `global_control` object affects, during its lifetime, the soft limit on the number of worker threads that a TBB scheduler can use. At any point in time, the soft limit is the minimum value of all of the `max_concurrency_limit` values requested by the active `global_control` objects. If the soft limit was initialized before any of the active `global_control` objects were constructed, this initial value is also considered when finding the minimum value. When a `global_control` object is destroyed, the soft limit may increase if the destroyed object was the limiting `max_concurrency_limit` value. Creation of a `global_control` object does not initialize the TBB scheduler nor increment the reference count on the scheduler. When the last `global_control` object is destroyed, the soft limit is reset to the default soft thread limit.

`task_scheduler_init` objects: A `task_scheduler_init` object creates the task arena associated with a master thread, but only if one does not already exist for that thread. If one already exists, the `task_scheduler_init` object increments the reference count of the task arena. When a `task_scheduler_init` object is destroyed, it decrements the reference count, and if the new count is zero, the task arena is destroyed. If a TBB scheduler does not exist when a `task_scheduler_init` object is constructed, a TBB scheduler is created, and if the soft thread limit has not been set by a `global_control` object, it is initialized using the constructor's `max_threads` argument shown as follows:

`P-1`, where `P` is the number of logical cores	if `max_threads` <= `P - 1`
`max_threads`	otherwise

`task_arena` objects: A `task_arena` object creates an explicit task arena that is not associated with a specific master thread. The underlying task arena is not initialized immediately during the constructor but lazily on first use (in our illustrations in this chapter, we show the construction of the object not the underlying task arena representation). If a thread spawns or enqueues a task into an explicit `task_arena` before that thread has initialized its own implicit task arena, this action acts like a first use of the TBB scheduler for that thread – including all of the side effects of a default initialization of its implicit task arena and possible initialization of the soft limit.

The Best Approaches for Setting the Number of Threads

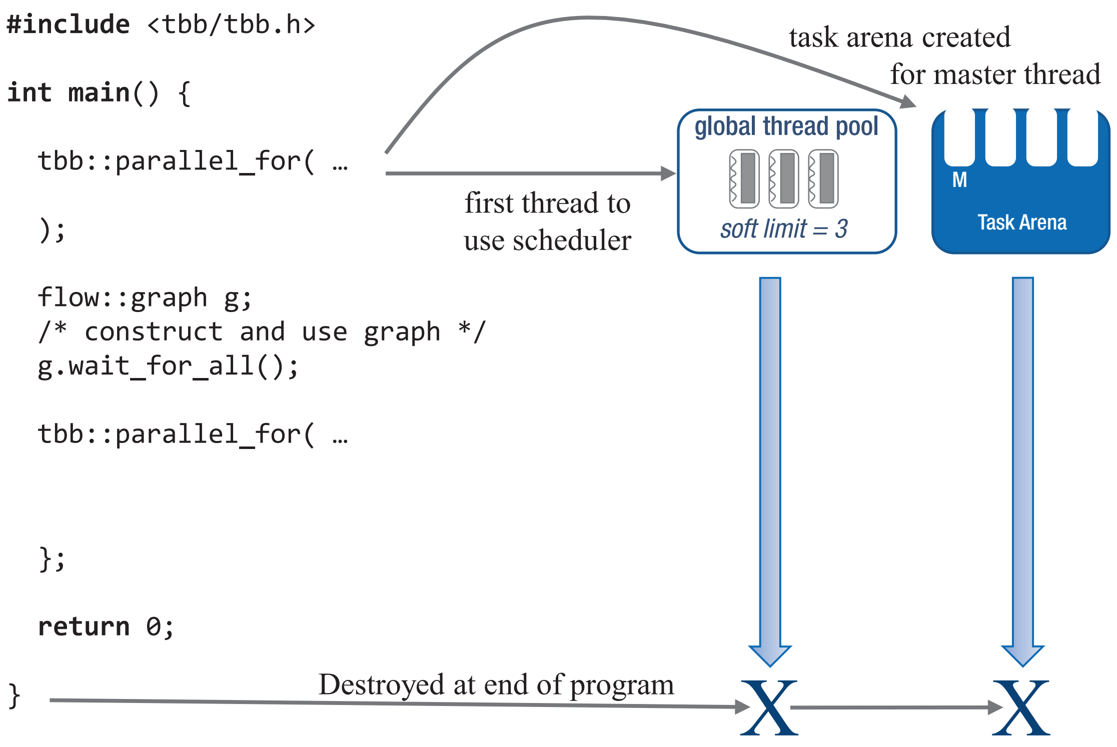
The combination of the `task_scheduler_init`, `task_arena`, and `global_control` classes provides a powerful set of tools for controlling the number of threads that can participate in the execution of parallel work in TBB.

The interaction of these objects can be confusing when combined in ways that fall outside of the expected patterns. Therefore, in this section, we focus on common scenarios and provide recommended approaches for using these classes. For simplicity in the figures that we show in this section, we assume that we are executing on a system that supports four logical cores. On such a system, the TBB library will, by default, create three worker threads, and there will be four slots in any default task arenas, with one slot reserved for a master thread. In our figures, we show the number of threads that are available in the global thread pool and the number of slots in the task arena(s). To reduce clutter in the figures, we do not show workers being assigned to slots. Downward arrows are used to indicate the lifetimes of objects. A large "X" indicates the destruction of an object.

Using a Single `task_scheduler_init` Object for a Simple Application

The simplest, and perhaps most common, scenario is that we have an application with a single main thread and no explicit task arenas. The application may have many TBB algorithms, including use of nested parallelism, but does not have more than one user-created thread – that is, the main thread. If we do nothing to control the number of threads managed by the TBB library, an implicit task arena will be created for the main thread when it first interacts with the TBB scheduler by spawning a task, executing a

TBB algorithm, or by using a TBB flow graph. When this default task arena is created, the global thread pool will be populated with one thread fewer than the number of logical cores in the system. This most basic case, with all default initializations, is illustrated for a system with four logical cores in Figure 11-5.



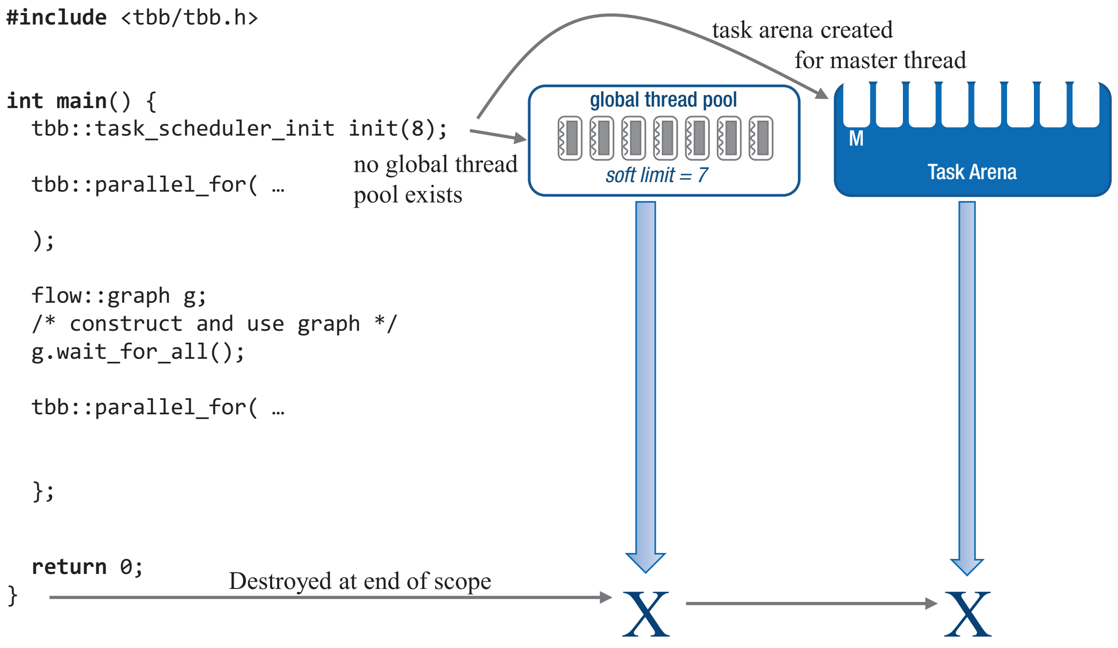
```
#include <tbb/tbb.h>

int main() {

  tbb::parallel_for( …

  );

  flow::graph g;
  /* construct and use graph */
  g.wait_for_all();

  tbb::parallel_for( …



  };

  return 0;

}
```

Figure 11-5. *Default initialization of the global thread pool and a single task arena for the main thread*

The sample code is available at Github in `ch11/fig_11_05.cpp` and is instrumented so that it prints a summary of how many threads participate in each section of the code. Many of the examples in this chapter are instrumented similarly. This instrumentation is not shown in the source code in the figures but is available in the code at Github. Running this example on a system with four logical cores results in output similar to

```
There are 4 logical cores.
4 threads participated in 1st pfor
4 threads participated in 2nd pfor
4 threads participated in flow graph
```

If we want different behavior in this simplest scenario, `class task_scheduler_init` is sufficient for controlling the number of threads. All we need to do is create a `task_scheduler_init` object before our first use of TBB tasks and pass to it the desired number of threads we want our application to use. An example of this is shown in Figure 11-6. The construction of this object creates the task scheduler, populates the global thread pool (market) with an appropriate number of threads (at least enough to fill the slots in the task arena[1]), and constructs a single arena for the main thread with the requested number of slots. This TBB scheduler is destroyed when the single `task_scheduler_init` object is destroyed.



```
#include <tbb/tbb.h>

int main() {
  tbb::task_scheduler_init init(8);

  tbb::parallel_for( …

  );

  flow::graph g;
  /* construct and use graph */
  g.wait_for_all();

  tbb::parallel_for( …

  };

  return 0;
}
```

Figure 11-6. *Using a single `task_scheduler_init` object for a simple application*

[1]This is a slight oversimplification. See the earlier sidebar on soft and hard limits in this chapter to learn more.

Executing the code for Figure 11-6 will result in an output:

```
There are 4 logical cores.
8 threads participated in 1st pfor
8 threads participated in 2nd pfor
8 threads participated in flow graph
```

Note Of course, statically coding the number of threads to use is a really bad idea. We are illustrating capabilities with easy to follow examples with specific numbers. In order to write portable and more timeless code, we would almost never recommend coding specific numbers.

Using More Than One `task_scheduler_init` Object in a Simple Application

A slightly more complicated use case is when we still have only a single application thread but we want to execute with different numbers of threads during different phases of the application. As long as we don't overlap the lifetimes of `task_scheduler_init` objects, we can change the number of threads during an application's execution by creating and destroying `task_scheduler_init` objects that use different `max_threads` values. A common scenario where this is used is in scaling experiments. Figure 11-7 shows a loop that runs a test on 1 through P threads. Here, we create and destroy a series of `task_scheduler_init` objects, and therefore TBB schedulers, that support different numbers of threads.

```
#include <iostream>
#include <tbb/tbb.h>

void run_test() {
  const int N =
    10*tbb::task_scheduler_init::default_num_threads();
  tbb::parallel_for(0, N, [](int) {
    tbb::tick_count t0 = tbb::tick_count::now();
    while ((tbb::tick_count::now() - t0).seconds() < 0.01);
  });
}

void fig_11_7() {
  const int P =
    tbb::task_scheduler_init::default_num_threads();
  for (int i = 1; i <= P; ++i) {
    tbb::tick_count t0 = tbb::tick_count::now();
    tbb::task_scheduler_init init(i);
    run_test();
    auto sec = (tbb::tick_count::now() - t0).seconds();
    std::cout << "Test using " << i << " threads took "
              << sec << "seconds" << std::endl;
  }
}
```

Figure 11-7. *A simple timing loop that runs a test using 1 through P threads*

In Figure 11-7, each time we create the `task_scheduler_init` object `init`, the library creates a task arena for the main thread with one slot reserved for a master thread and `i-1` additional slots. At the same time, it sets the soft limit and populates the global thread pool with at least `i-1` worker threads (remember that that if `max_threads` is < `P-1`, it still creates `P-1` threads in the global thread pool). When `init` is destroyed, the TBB scheduler is destroyed, including the single task arena and the global thread pool.

The output from a run of the sample code, in which `run_test()` contains a `parallel_for` with 400 milliseconds of work, results in output similar to

```
Test using 1 threads took 0.401094seconds
Test using 2 threads took 0.200297seconds
Test using 3 threads took 0.140212seconds
Test using 4 threads took 0.100435seconds
```

Using Multiple Arenas with Different Numbers of Slots to Influence Where TBB Places Its Worker Threads

Let's now explore even more complicated scenarios, where we have more than one task arena. The most common way this situation arises is that our application has more than one application thread. Each of these threads is a master thread and gets its own implicit task arena. We can also have more than one task arena because we explicitly create arenas using `class task_arena` as described in Chapter 12. Regardless of how we wind up with multiple task arenas in an application, the worker threads migrate to task arenas in proportion to the number of slots they have. And the threads only consider task arenas that have tasks available to execute. As we noted earlier, we are assuming in this chapter that tasks are all of equal priority. Task priorities, which can affect how threads migrate to arenas, are described in more detail in Chapter 14.

Figure 11-8 shows an example with a total of three task arenas: two task arenas that are created for master threads (the main thread and thread `t`) and one explicit task arena, `a`. This example is contrived but shows code that is complicated enough to get our points across.

In Figure 11-8, there is no attempt to control the number of threads in the application or the number of slots in the task arenas. Therefore, each arena is constructed with the default number of slots, and the global thread pool is initialized with the default number of worker threads as shown in Figure 11-9.

```
void fig_11_8() {
  tbb::task_arena a;

  std::thread t([=]() {
    tbb::parallel_for(0, N, [](int) { /* do work */ });
  });

  a.execute([=]() { tbb::parallel_for(0, N,
    [](int) { /* do work */}
  );

  tbb::parallel_for(0, N, [](int) { /* do work */ });
  t.join();
}
```

Figure 11-8. *An application with three task arenas: the default arena for the main thread, an explicit `task_arena a`, and a default task arena for master thread `t`*

Because we now have more than one thread, we use the vertical position in Figure 11-9 to indicate time; objects lower in the figure are constructed after objects higher in the figure. The figure shows one possible execution order, and in our illustration thread `t` is the first thread to spawn a task, by using a `parallel_for`, and so it creates the TBB scheduler and the global thread pool. As complicated as the example appears, the behavior is well defined.

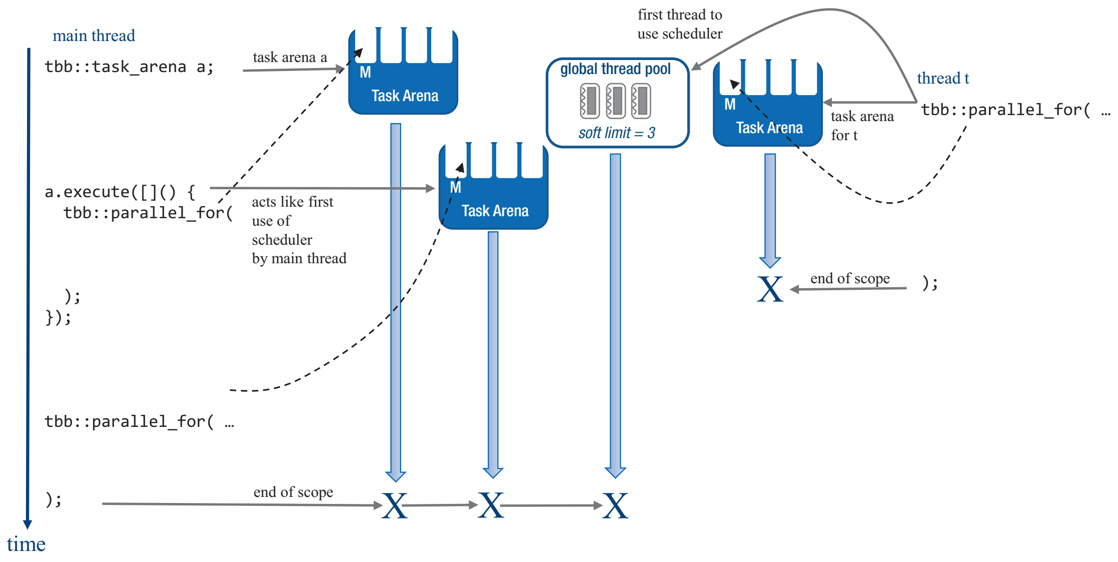


Figure 11-9. A possible execution of the example with three task arenas

As shown in Figure 11-9, the execution of the `parallel_for` algorithms in thread `t` and task arena `a` might overlap. If so, the three threads in the global thread pool are divided between them. Since there are three worker threads, one arena will initially get one worker thread and the other one will initially get two worker threads. Which arena gets fewer threads is up to the library's discretion, and when either of these arenas runs out of work, the threads can migrate to the other arena to help finish the remaining work there. After the call to `a.execute` completes in the main thread in Figure 11-9, the final `parallel_for` executes within the main thread's default arena, with the main thread filling its master slot. If at this point, the `parallel_for` in thread `t` is also complete, then all three worker threads can migrate to the main thread's arena to work on the final algorithm.

The default behavior shown in Figure 11-9 makes a lot of sense. We only have four logical cores in our system, so TBB initializes the global thread pool with three threads. When each task arena is created, TBB doesn't add more threads to the global thread pool because the platform still has the same number of cores. Instead, the three threads in the global thread pool are dynamically shared among the task arenas.

The TBB library assigns threads to task arenas in proportion to the number of slots they have. But we don't have to settle for task arenas with the default number of slots. We can control the number of slots in the different arenas by creating a `task_scheduler_init` object for each application thread and/or by passing in a `max_concurrency` argument to explicit `task_arena` objects. A modified example that does this is shown in Figure 11-10.

```
void fig_11_10() {
  int N = 10*P;
  tbb::task_scheduler_init i4(4);

  tbb::task_arena a(3);

  std::thread t([=]() {
    tbb::task_scheduler_init i2(2);
    tbb::parallel_for(0, N, [](int) { /* do work */ });
  });

  a.execute([=]() {
    tbb::parallel_for(0, N, [](int) { /* do work */ });
  });

  tbb::parallel_for(0, N, [](int) { /* do work */ });
  t.join();
}
```

Figure 11-10. *An application with three task arenas: the default arena for the main thread will have a max concurrency of 4, the explicit `task_arena` a has a max concurrency of 3, and the default arena for master thread t has a max concurrency of 2.*

Now when we execute the application, the TBB library will only be able to provide at most one worker thread to thread `t`'s arena since it only has a single slot for a worker, and the remaining two can be assigned to the `parallel_for` in arena `a`. We can see an example execution that shows this in Figure 11-11.

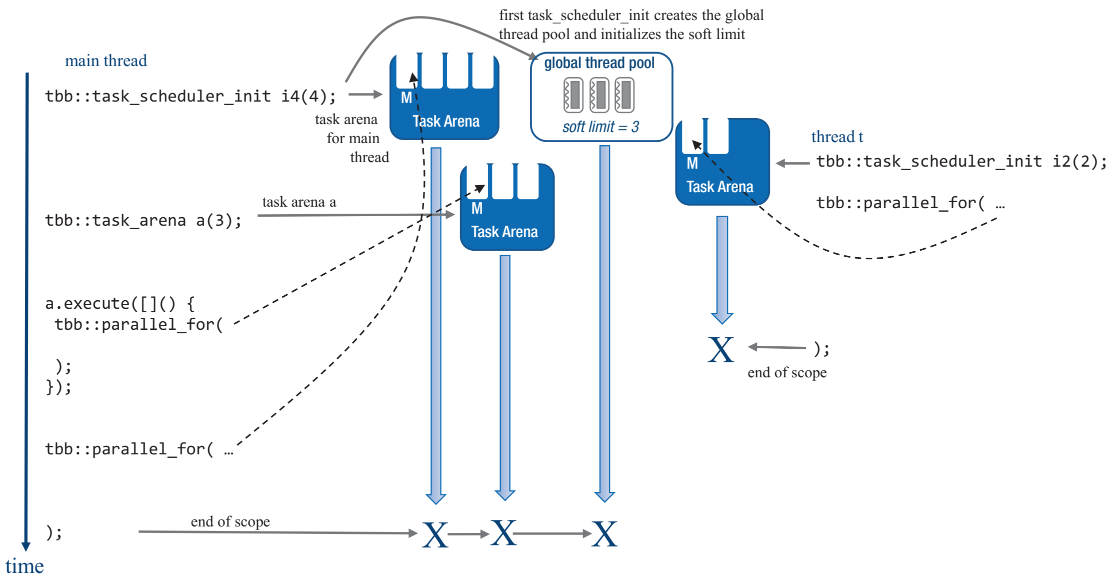


Figure 11-11. *A possible execution of the example with three task arenas after we have explicitly set the number of slots in the various arenas*

An execution of the sample code from Github, which tracks how many threads participates in each section, shows an output of

```
There are 4 logical cores.
3 threads participated in arena pfor
4 threads participated in main pfor
2 threads participated in std::thread pfor
```

Because we have limited the number of slots available to thread `t`, the other threads can no longer migrate from `task_arena a` to thread `t` after they finish their work. We need to be prudent when we limit slots. In this simple example, we have skewed execution in favor of `task_arena a` but have also restricted how many idle threads can assist thread `t`.

We have now controlled the number of slots for threads in task arenas but still relied on the default number of threads that TBB allocates in the global thread pool to fill these slots. If we want to change the number of threads that are available to the fill the slots, we need to turn to the `class global_control`.

Using `global_control` to Control How Many Threads Are Available to Fill Arena Slots

Let's revisit the example from the previous section one more time, but double the number of threads in the global thread pool. Our new implementation is shown in Figure 11-12.

```
void fig_11_12() {
  const int P =
tbb::task_scheduler_init::default_num_threads();
  int N = 10*P;

  auto mp = tbb::global_control::max_allowed_parallelism;
  int nt = 2*P;
  tbb::global_control gc(mp, nt);

  tbb::task_arena a(3*nt/4);

  std::thread t([=]() {
    tbb::task_scheduler_init i1(nt/4);
    tbb::parallel_for(0, N, [](int) { /* do work */ });
  });

  a.execute([=]() {
    tbb::parallel_for(0, N, [](int) { /* do work */ });
  });

  tbb::parallel_for(0, N, [](int) { /* do work */ });
  t.join();
}
```

Figure 11-12. *An application with three task arenas and a `global_control` object*

We now use a `global_control` object to set the number of threads in the global thread pool. Remember that a `global_control` object is used to affect global parameters used by the scheduler; in this case, we are changing the `max_allowed_parallelism` parameter. We also use a `task_scheduler_init` object in thread `t` and an argument to the `task_arena` constructor to set the maximum number of threads that can be assigned to each task arena. Figure 11-13 shows an example execution on our four-core machine. The application now creates seven worker threads (eight total threads minus the already available master thread), and the worker threads are divided up unequally between the explicit `task_arena a` and the default arena for thread `t`. Since we do nothing special for the main thread, the final `parallel_for` uses its default task arena with P slots.

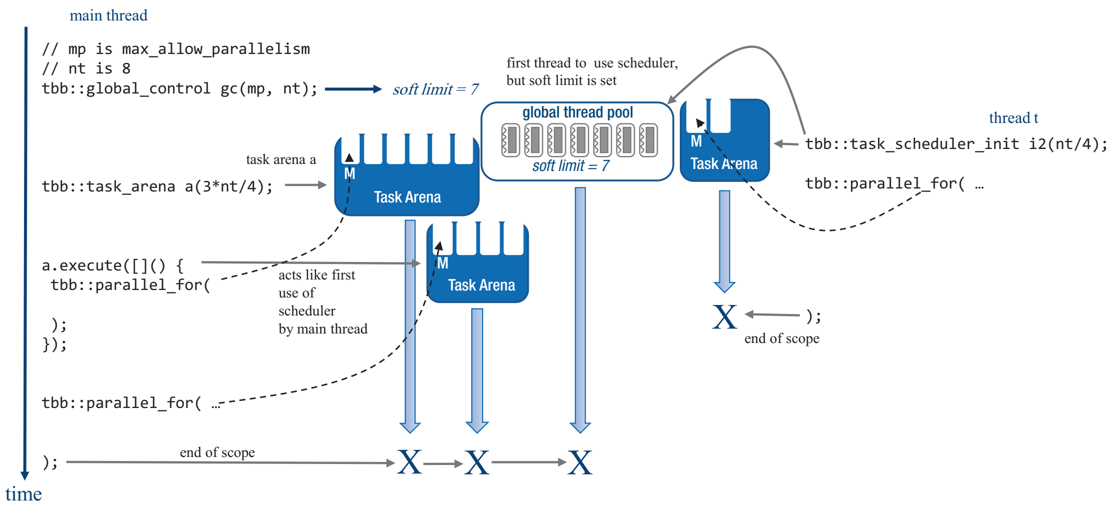

main thread
// mp is max_allow_parallelism
// nt is 8
tbb::global_control gc(mp, nt);
soft limit = 7
first thread to use scheduler, but soft limit is set
global thread pool
soft limit = 7
thread t
tbb::task_scheduler_init i2(nt/4);
tbb::parallel_for(…
task arena a
tbb::task_arena a(3*nt/4);
M
Task Arena
a.execute([]() {
tbb::parallel_for(
);
});
acts like first use of scheduler by main thread
tbb::parallel_for(…
);
end of scope
X
time

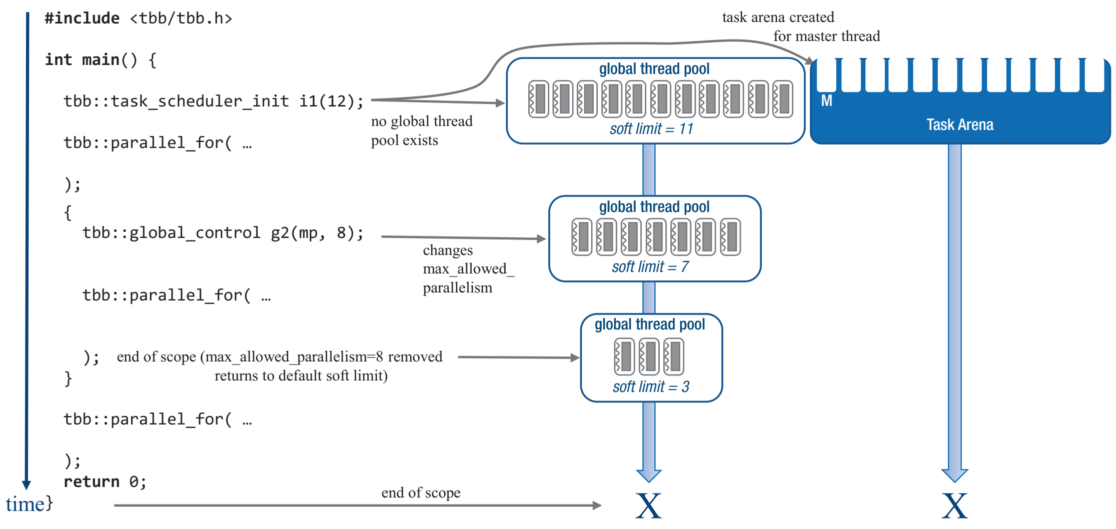
Figure 11-13. *A possible execution of the example with three task arenas after we have explicitly set the soft limit using a `global_control` object*

Executing the sample code for Figure 11-13 yields an output similar to

```
There are 4 logical cores.
6 threads participated in arena pfor
4 threads participated in main pfor
2 threads participated in std::thread pfor
```

Using `global_control` to Temporarily Restrict the Number of Available Threads

Another common scenario is to use a `global_control` object to temporarily change the number of threads for a specific phase of an application as shown in Figure 11-14. In this example, the master thread creates a thread pool and task arena that can support 12 worker threads by constructing a `task_scheduler_init` object. But a `global_control` object is used to restrict a specific `parallel_for` to only seven worker threads. While the task arena retains 12 slots during the whole application, the number of threads available in the thread pool is temporarily reduced, so at most seven of the slots in the task arena can be filled with workers.



```
#include <tbb/tbb.h>

int main() {

  tbb::task_scheduler_init i1(12);

  tbb::parallel_for( …

  );
  {
    tbb::global_control g2(mp, 8);


    tbb::parallel_for( …


    );
  }

  tbb::parallel_for( …

  );
  return 0;
}
```

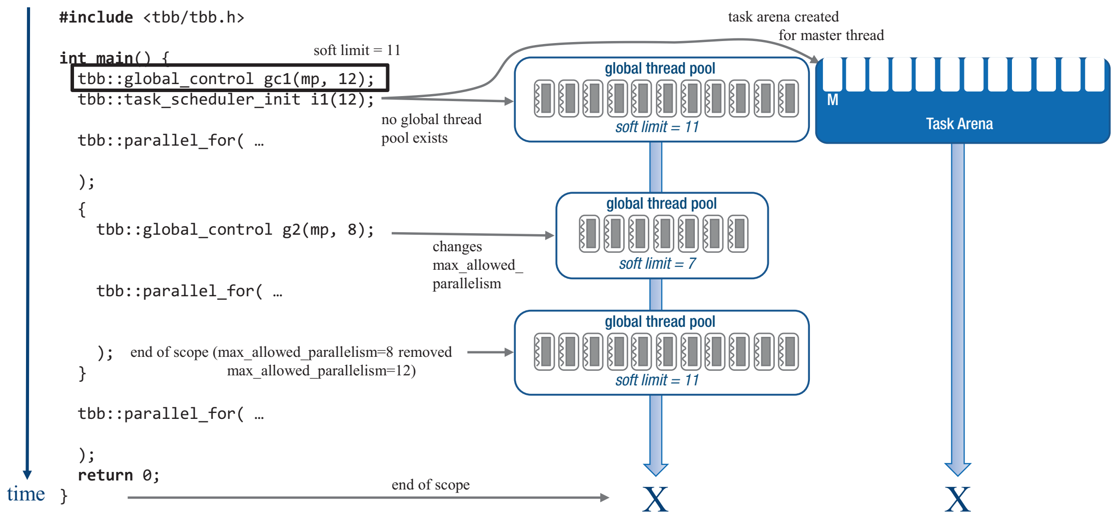
Figure 11-14. *Using a `global_control` object to temporarily change the number of threads available for a specific algorithm instance and then return to default setting*

When the `global_control` object is destroyed, the soft limit is recalculated, using any remaining `global_control` objects. Since there are none, the soft limit is set to the default soft limit. This perhaps unexpected behavior is important to note, since we need to create an outer `global_control` object if we want to maintain 11 threads in the global thread pool. We show this in Figure 11-15.

In Figures 11-14 and 11-15, we cannot use a `task_scheduler_init` object to temporarily change the number of threads because a task arena already exists for the main thread. If we create another `task_scheduler_init` object in the inner scope, it only increments the reference count on that task arena and does not create a new one. Therefore, we use a `global_control` object to restrict the number of threads that are available instead of reducing the number of arena slots.

If we execute the code in Figure 11-14, we see an output similar to

```
There are 4 logical cores.
12 threads participated in 1st pfor
8 threads participated in 2nd pfor
4 threads participated in 3rd pfor
```



```
#include <tbb/tbb.h>

int main() {
  tbb::global_control gc1(mp, 12);
  tbb::task_scheduler_init i1(12);

  tbb::parallel_for( …

  );
  {
    tbb::global_control g2(mp, 8);


    tbb::parallel_for( …


    );
  }

  tbb::parallel_for( …

  );
  return 0;
}
```

time
soft limit = 11
task arena created for master thread
no global thread pool exists
global thread pool
soft limit = 11
M
Task Arena
changes max_allowed_ parallelism
global thread pool
soft limit = 7
end of scope (max_allowed_parallelism=8 removed max_allowed_parallelism=12)
global thread pool
soft limit = 11
end of scope
X
X

Figure 11-15. *Using* `global_control` *objects to temporarily change the number of threads available for a specific algorithm instance*

After adding an outer `global_control` object, as done in Figure 11-15, the resulting output is

```
There are 4 logical cores.
12 threads participated in 1st pfor
8 threads participated in 2nd pfor
12 threads participated in 3rd pfor
```

When NOT to Control the Number of Threads

When implementing a plugin or a library, its best to avoid using `global_control` objects. These objects affect global parameters, so our plugin or library function will change the number of threads available to all of the components in the application. Given the local view of a plugin or library, that's probably not something it should do. In Figure 11-14, we temporarily changed the number of threads in the global thread pool. If we did something like this from inside a library call, it would not only affect the number of threads available in the task arena of the calling thread, but every task arena in our application. How can a library function know this is the right thing to do? It very likely cannot.

We recommend that libraries do not meddle with global parameters and leave that only to the main program. Developers of applications that allow plugins should clearly communicate to plugin writers what the parallel execution strategy of the application is, so that they can implement their plugins appropriately.

SETTING THE STACK SIZE FOR WORKER THREADS

The `task_scheduler_init` and `global_control` classes can also be used to set the stack size for the worker threads. The interaction of multiple objects are the same as when used to set the number of threads, with one exception. When there is more than one `global_control` object that sets the stack size, the stack size is the *maximum*, not the minimum, of the requested values.

The second argument to the `task_scheduler_init` object is `thread_stack_size`. A value of 0, which is the default, instructs the scheduler to use the default for that platform. Otherwise, the provided value is used.

The `global_control` constructor accepts a parameter and value. If the parameter argument is `thread_stack_size`, then the object changes the value for the global stack size parameter. Unlike the `max_allowed_paralleism` value, the global `thread_stack_size` value is the maximum of the requested values.

Why change the default stack size?

A thread's stack has to be large enough for all of memory that is allocated on its stack, including all of the local variables on its call stack. When deciding how much stack is needed, we have to consider the local variables in our task bodies but also how recursive execution of task trees might lead to deep recursion, especially if we have implemented our own task-based algorithms using task blocking. If we don't remember how this style can lead to an explosion in stack usage, we can look back at the section, **The low-level task interface: part one/task blocking** in Chapter 10.

Since the proper stack size is application dependent, there is unfortunately no good rule of thumb to share. TBB's OS-specific default is already a best guess at what a thread typically needs.

Figuring Out What's Gone Wrong

The `task_scheduler_init`, `task_arena`, and `global_control` classes were introduced over time into the TBB library to solve specific problems. The `task_scheduler_init` class was sufficient in the early days of TBB, when few applications were parallel, and when they were, there was often only a single application thread. The `task_arena` class helped users manage isolation in applications as they became more complex. And the `global_control` class gave users better control of the global parameters used by the library to further manage complexity. Unfortunately, these features were not created together as part of one cohesive design. The result is that when used outside of the scenarios we have previously outlined, their behaviors can sometimes be nonintuitive, even if they are well defined.

The two most common sources of confusion are (1) knowing when a TBB scheduler is created by default and (2) races to set the global thread pool's soft limit.

If we create a `task_scheduler_init` object it either creates a TBB scheduler or else increments the reference count on the scheduler if it already exists. Which interfaces in the TBB library act like a first use of the TBB scheduler can be hard to keep straight. It's very clear that executing any of the TBB algorithms, using a TBB flow graph or spawning tasks, is a use of the TBB scheduler. But as we noted early, even executing tasks in an explicit `task_arena` is treated as a first use of the TBB scheduler, which impacts not only the explicit task arena, but may impact the calling thread's default task arena. What about using thread local storage or using one of the concurrent containers? These do not count. The best advice, other than paying close attention to the implications of the interfaces being used, is that if an application uses an unexpected number of threads – especially if it uses the default number of threads when you think you have changed the default – is to look for places where a default TBB scheduler may have been inadvertently initialized.

The second common cause of confusion is races to set the soft limit on the number of available threads. For example, if two application threads execute in parallel and both create a `task_scheduler_init` object, the first one to create its object will set the soft limit. In Figure 11-16, two threads executing concurrently in the same application both create `task_scheduler_init` objects – one requesting `max_threads=4` and the other `max_threads=8`. What happens with the task arenas is simple: each master thread gets its own task arena with the number of slots it requested. But what if the soft limit on the number of threads in the global thread pool has not been set yet? How many threads does the TBB library populate the global thread pool with? Should it create `3` or `7` or `3+7=10` or `P-1` or ...?

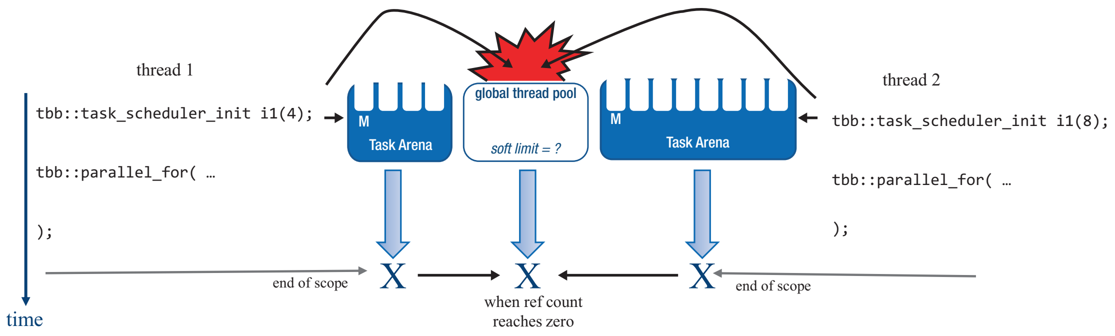


***Figure 11-16.** The concurrent use of two `task_scheduler_init` objects*

As we outlined in our description of `task_scheduler_init`, it does none of these things. Instead, it uses whichever request comes first. Yes, you read that right! If thread 1 just so happens to create its `task_scheduler_init` object first, we get a TBB scheduler with a global thread pool with three worker threads. If thread 2 creates its `task_scheduler_init` object first, we get a thread pool with seven worker threads. Our two threads may be sharing three worker threads or seven worker threads; it all depends on which one wins the race to create the TBB scheduler first!

We shouldn't despair though; almost all of the potential pitfalls that come along with setting the number of threads can be addressed by falling back to the common usage patterns described earlier in this chapter. For example, if we know that our application may have a race like that shown in Figure 11-16, we can make our desires crystal clear by setting the soft limit in the main thread using a `global_control` object.

Summary

In this chapter, we provided a brief recap of the structure of the TBB scheduler before introducing the three classes used to control the number of threads used for parallel execution: `class task_scheduler_init`, `class task_arena`, and `class global_control`. We then described common use cases for controlling the number of threads used by parallel algorithms – working from simple cases where there is a single main thread and a single task arena to more complex cases where there are multiple master threads and multiple task arenas. We concluded by pointing out that while there are potential gotchas in using these classes, we can avoid these by carefully using the classes to make our intention clear without relying on default behaviors or the winners of races.