CHAPTER 13

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

When developing parallel applications with the Threading Building Blocks library, we create tasks by using the high-level execution interfaces or the low-level APIs. These tasks are scheduled by the TBB library onto software threads using work stealing. These software threads are scheduled by the Operating System (OS) onto the platform's cores (hardware threads). In this chapter, we discuss the features in TBB that let us influence the scheduling choices made by the OS and by TBB. *Thread-to-core affinity* is used when we want to influence the OS so that it schedules the software threads onto particular core(s). *Task-to-thread affinity* is used when we want to influence the TBB scheduler so that it schedules tasks onto particular software threads. Depending on what we are trying to achieve, we may be interested in one kind of affinity or the other, or a combination of both.

There can be different motivations for creating affinity. One of the most common motivations is to take advantage of data locality. As we have repeatedly noted in this book, data locality can have a huge impact on the performance of a parallel application. The TBB library, its high-level execution interfaces, its work-stealing scheduler, and its concurrent containers have all been designed with locality in mind. For many applications, using these features will lead to good performance without any manual tuning. Sometimes though, we will need to provide hints or take matters completely into our own hands so that the schedulers, in TBB and the OS, more optimally schedule work near its data. In addition to data locality, we might also be interested in affinity when using heterogeneous systems, where the capabilities of cores differ, or when software threads have different properties, such as higher or lower priorities.

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_13

In Chapter 16, the high-level features for data locality that are exposed by the TBB parallel algorithms are presented. In Chapter 17, the features for tuning cache and memory use in TBB flow graphs are discussed. In Chapter 20, we showed how to use features of the TBB library to tune for Non-Uniform Memory Access (NUMA) architectures. For many readers, the information in those chapters will be sufficient to accomplish the specific tasks they need to perform to tune their applications. In this chapter, we focus on the lower-level, fundamental support provided by the TBB's scheduler and tasks that are sometimes abstracted by the high-level features described in those chapters or sometimes used directly in those chapters to create affinity.

Creating Thread-to-Core Affinity

All of the major operating systems provide interfaces that allow users to set the affinity of software threads, including `pthread_setaffinity_np` or `sched_setaffinity` on Linux and `SetThreadAffinityMask` on Windows. In Chapter 20, we use the Portable Hardware Locality (hwloc) package as a portable way to set affinity across platforms. In this chapter, we do not focus on the mechanics of setting affinity – since these mechanics will vary from system to system – instead we focus on the hooks provided by the TBB library that allow us to use these interfaces to set affinity for TBB master and worker threads.

The TBB library by default creates enough worker threads to match the number of available cores. In Chapter 11, we discussed how we can change those defaults. Whether we use the defaults or not, the TBB library does not automatically affinitize these threads to specific cores. TBB allows the OS to schedule and migrate the threads as it sees fit. Giving the OS flexibility in where it places TBB threads is an intentional design choice in the library. In a multiprogrammed environment, an environment in which TBB excels, the OS has visibility of all of the applications and threads. If we make decisions about where threads should execute from within our limited view inside of a single application, we might make choices that lead to poor overall system resource utilization. Therefore, it is often better to not affinitize threads to cores and instead allow the OS to choose where the TBB master and worker threads execute, including allowing it to dynamically migrate threads during a program's execution.

However, like we will see in many chapters of this book, the TBB library provides features that let us change this behavior if we wish. If we want to force TBB threads to have affinity for cores, we can use the `task_scheduler_observer` class to do so (see **Observing the scheduler with the `task_scheduler_observer` class**). This class lets an application define callbacks that are invoked whenever a thread enters and leaves the TBB scheduler, or a specific task arena, and use these callbacks to assign affinity. The TBB library does not provide an abstraction to assist with making the OS-specific calls required to set thread affinity, so we have to handle these low-level details ourselves using one of the OS-specific or portable interfaces we mentioned earlier.

OBSERVING THE SCHEDULER WITH THE `TASK_SCHEDULER_OBSERVER` CLASS

The `task_scheduler_observer` class provides a way to observe when a thread starts or stops participating in task scheduling. The interface of this class is shown as follows:

```
class task_scheduler_observer {
public:
  task_scheduler_observer();
  virtual ~task_scheduler_observer();
  void observe( bool state=true );
  bool is_observing() const;
  virtual void on_scheduler_entry( bool is_worker ) {}
  virtual void on_scheduler_exit( bool is_worker } {}
};
```

To use the class, we create our own class that inherits from `task_scheduler_observer` and implements the `on_scheduler_entry` and `on_scheduler_exit` callbacks. When an instance of this class is constructed and its `observe` state is set to true, the entry and exit functions will be called whenever a master or worker thread enters or exits the global TBB task scheduler.

A recent extension to the class now allows us to pass a `task_arena` to the constructor. This extension was a preview feature prior to TBB 2019 Update 4 but is now fully supported. When a `task_arena` reference is passed, the observer will only receive callbacks for threads that enter and exit that specific arena:

```
explicit task_scheduler_observer(task_arena & a);
```

Figure 13-1 shows a simple example of how to use a `task_scheduler_observer` object to pin threads to cores on Linux. In this example, we use the `sched_setaffinity` function to set the CPU mask for each thread as it joins the default arena. In Chapter 20, we show an example that assigns affinity using the hwloc software package. In the example in Figure 13-1, we use `tbb::this_task_arena::max_concurrency()` to find the number of slots in the arena and `tbb::this_task_arena::current_thread_index()` to find the slot that the calling thread is assigned to. Since we know there will be the same number of slots in the default arena as the number of logical cores, we pin each thread to the logical core that matches its slot number.

```
#include <iostream>
#include <sched.h>
#define TBB_PREVIEW_LOCAL_OBSERVER 1
#include <tbb/tbb.h>

thread_local int my_cpu = -1;
void doWork();

class pinning_observer : public tbb::task_scheduler_observer {
public:
    pinning_observer() { observe(true); }

    void on_scheduler_entry( bool is_worker ) {
        cpu_set_t *mask;
        auto number_of_slots =
          tbb::this_task_arena::max_concurrency();
        mask = CPU_ALLOC(number_of_slots);
        auto mask_size = CPU_ALLOC_SIZE(number_of_slots);

        auto slot_number =
          tbb::this_task_arena::current_thread_index();
        CPU_ZERO_S(mask_size, mask);
        CPU_SET_S(slot_number, mask_size, mask);
        if (sched_setaffinity( 0, mask_size, mask )) {
           std::cout << "Error in sched_setaffinity"
                     << std::endl;
        }
        // so we can see if it worked:
        my_cpu = sched_getcpu();
    }
};

void fig_13_1() {
  const int N = 100;

  std::cout << "Without pinning:" << std::endl;
  tbb::parallel_for(0, N, [](int) {doWork();});

  std::cout << std::endl
            << "With pinning:" << std::endl;
  pinning_observer p;
  tbb::parallel_for(0, N, [](int) {doWork();});
  std::cout << std::endl;
}
```

Figure 13-1. *Using a* `task_scheduler_observer` *to pin threads to cores on a Linux platform*

We can of course create more complicated schemes for assigning logical cores to threads. And, although we don't do this in Figure 13-1, we can also store the original CPU mask for each thread so that we can restore it when the thread leaves the arena.

As we discuss in Chapter 20, we can use the `task_scheduler_observer` class, combined with explicit `task_arena` instances, to create isolated groups of threads that are restricted to the cores that share the same local memory banks in a Non-Uniform-Memory Access (NUMA) system, a NUMA node. If we also control data placement, we can greatly improve performance by spawning the work into the arena of the NUMA node on which its data resides. See Chapter 20 for more details.

We should always remember that if we use thread-to-core affinity, we are preventing the OS from migrating threads away from oversubscribed cores to less-used cores as it attempts to optimize system utilization. If we do this in production applications, we need to be sure that we will not degrade multiprogrammed performance! As we'll mention several more times, only systems dedicated to running a single application (at a time) are likely to have an environment in which limiting dynamic migration can be of benefit.

Creating Task-to-Thread Affinity

Since we express our parallel work in TBB using tasks, creating thread-to-core affinity, as we described in the previous section, is only one part of the puzzle. We may not get much benefit if we pin our threads to cores, but then let our tasks get randomly moved around by work stealing!

When using the low-level TBB tasking interfaces introduced in Chapter 10, we can provide hints that tell the TBB scheduler that it should execute a task on the thread in a particular arena slot. Since we will likely use the higher-level algorithms and tasking interfaces whenever possible, such as `parallel_for`, `task_group` and flow graphs, we will rarely use these low-level interfaces directly however. Chapter 16 shows how the `affinity_partitioner` and `static_partitioner` classes can be used with the TBB loop algorithms to create affinity without resorting to these low-level interfaces. Similarly, Chapter 17 discusses the features of TBB flow graphs that affect affinity.

So while task-to-thread affinity is exposed in the low-level task class, we will almost exclusively use this feature through high-level abstractions. Therefore using the interfaces we describe in this section is reserved for TBB experts that are writing their own algorithms using the lowest-level tasking interfaces. If you're such an expert, or

want to have a deeper understanding of how the higher-level interfaces achieve affinity, keep reading this section.

Figure 13-2 shows the functions and types provided by the TBB `task` class that we use to provide affinity hints.

```
typedef implementation-defined-unsigned-type affinity_id;
virtual void note_affinity( affinity_id id );
void set_affinity( affinity_id id );
affinity_id affinity() const;
```

Figure 13-2. *The functions in `tbb::task` that are used for task to thread affinity*

The type `affinity_id` is used to represent the slot in an arena that a task has affinity for. A value of zero means the task has no affinity. A nonzero value has an implementation-defined value that maps to an arena slot. We can set the affinity of task to an arena slot before spawning it by passing an `affinity_id` to its `set_affinity` function. But since the meaning of `affinity_id` is implementation defined, we don't pass a specific value, for example 2 to mean slot 2. Instead, we capture an `affinity_id` from a previous task execution by overriding the `note_affinity` callback function.

The function `note_affinity` is called by the TBB library before it invokes a task's `execute` function when (1) the task has no affinity but will execute on a thread other than the one that spawned it or (2) the task has affinity but it will execute on a thread different than the one specified by its affinity. By overriding this callback, we can track TBB stealing behavior so we can provide hints to the library to recreate this same stealing behavior in a subsequent execution of the algorithm, as we will see in the next example.

Finally, the `affinity` function lets us query a task's current affinity setting.

Figure 13-3 shows a class that inherits from `tbb::task` and uses the task affinity functions to record `affinity_id` values into a global array `a`. It only records the value when its `doMakeNotes` variable is set to true. The `execute` function prints the task id, the slot of the thread it is executing on, and the value that was recorded in the array for this task id. It prefixes its reporting with "hmm" if the task's `doMakeNotes` is true (it will then record the value), "yay!" if the task is executing in the arena slot that was recorded in array `a` (it was scheduled onto the same thread again), and "boo!" if it is executing in a different arena slot. The details of the printing are contained in the function `printExclaim`.

```
#include <iostream>
#define TBB_PREVIEW_LOCAL_OBSERVER 1
#include <tbb/tbb.h>

const int numTasks = 8;
tbb::task::affinity_id a[numTasks];
void doWork();
void printExclaim(const std::string &str, int id,
                  int slot, int note);

class MyTask : public tbb::task {
  int taskId;
  bool doMakeNotes;
public:
  MyTask(int id, bool do_make_notes = true)
    : taskId(id), doMakeNotes(do_make_notes) { }

  void note_affinity(tbb::task::affinity_id id) override {
    if (doMakeNotes) a[taskId] = id;
  }

  tbb::task *execute() override {
    auto slot_number = tbb::this_task_arena::current_thread_index();
    tbb::task::affinity_id a_id = a[taskId];
    std::string exclaim = "yay!";
    if (a_id != 0 && slot_number != a_id-1) exclaim = "boo!";
    if (doMakeNotes) exclaim = "hmm.";
    printExclaim(exclaim, taskId, slot_number, a_id);
    doWork();
    return NULL;
  }
};

void executeTaskTree(const std::string &name, bool note_affinity,
                     bool set_affinity) {
  std::cout << name << std::endl << "id:slot:a[i]" << std::endl;
  tbb::task *root = new(tbb::task::allocate_root()) tbb::empty_task;
  root->set_ref_count(numTasks+1);
  for (int i = 0; i < numTasks; ++i) {
    tbb::task *t = new (root->allocate_child())
                     MyTask(i,note_affinity);
    if (set_affinity && a[i]) t->set_affinity(a[i]);
    tbb::task::spawn(*t);
  }
  root->wait_for_all();
}

void fig_13_3() {
  std::fill(a, a+numTasks, 0);
  executeTaskTree("note_affinity", true, false);
  executeTaskTree("without set_affinity", false, false);
  executeTaskTree("with set_affinity", false, true);
}
```

Figure 13-3. *Using the task affinity functions*

While the meaning of `affinity_id` is implementation defined, TBB is open source, and so we peaked at the implementation. We therefore know that the `affinity_id` is 0 if there is no affinity, but otherwise it is the slot index plus 1. We should not depend on this knowledge in production uses of TBB, but we depend on it in our example's `execute` function so we can assign the correct exclamation "yay!" or "boo!".

The function `fig_13_3` in Figure 13-3 builds and executes three task trees, each with eight tasks, and assigns them ids from 0 to 7. This sample uses the low-level tasking interfaces we introduced in Chapter 10. The first task tree uses `note_affinity` to track when a task has been stolen to execute on some other thread than the master. The second task tree executes without noting or setting affinities. Finally, the last task tree uses `set_affinity` to recreate the scheduling recorded during the first run.

When we executed this example on a platform with eight threads, we recorded the following output:

```
note_affinity
id:slot:a[i]
hmm. 7:0:-1
hmm. 0:1:1
hmm. 1:6:6
hmm. 2:3:3
hmm. 3:2:2
hmm. 4:4:4
hmm. 5:7:7
hmm. 6:5:5

without set_affinity
id:slot:a[i]
yay! 7:0:-1
boo! 0:4:1
boo! 1:3:6
boo! 4:5:4
boo! 3:7:2
boo! 2:2:3
boo! 5:6:7
boo! 6:1:5
```

```
with set_affinity
id:slot:a[i]
yay! 7:0:-1
yay! 0:1:1
yay! 4:4:4
yay! 5:7:7
yay! 2:3:3
yay! 3:2:2
yay! 6:5:5
yay! 1:6:6
```

From this output, we see that the tasks in the first tree are distributed over the eight available threads, and the `affinity_id` for each task is recorded in array `a`. When the next set of tasks is executed, the recorded `affinity_id` for each task is not used to set affinity, and the tasks are randomly stolen by different threads. This is what random stealing does! But, when we execute the final task tree and use `set_affinity`, the thread assignments from the first run are repeated. Great, this worked out exactly as we wanted!

However, `set_affinity` only provides an affinity *hint* and the TBB library is actually free to ignore our request. When we set affinity using these interfaces, a reference to the task-with-affinity is placed in the targeted thread's affinity mailbox (see Figure 13-4). But the actual task remains in the local deque of the thread that spawned it. The task dispatcher only checks the affinity mailbox when it runs out of work in its local deque, as shown in the task dispatch loop in Chapter 9. So, if a thread does not check its affinity mailbox quickly enough, another thread may steal or execute its tasks first.

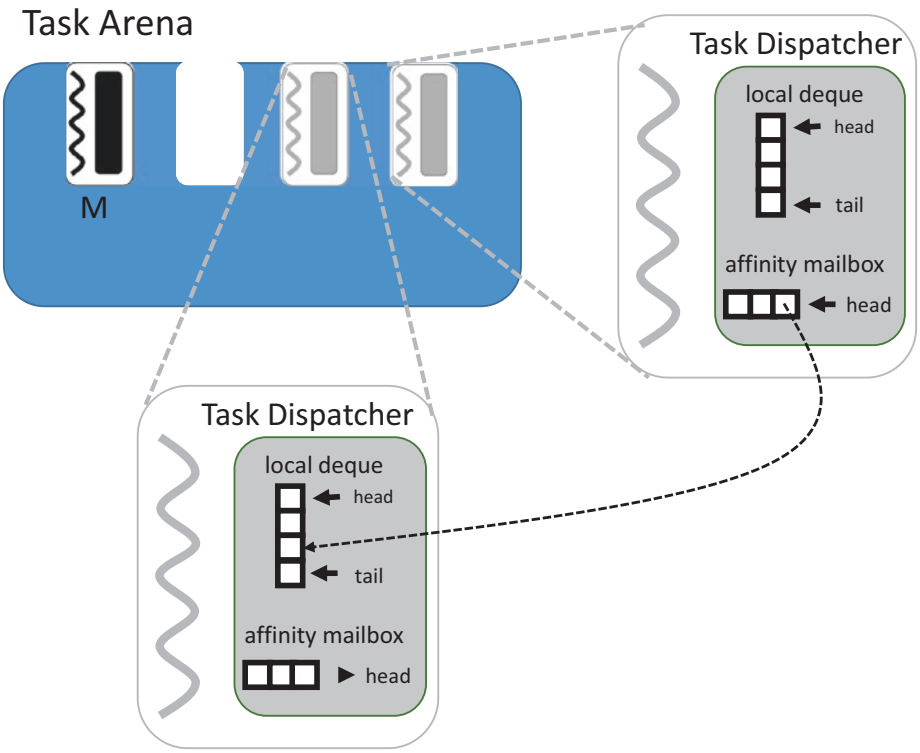

Figure 13-4. *The affinity mailbox holds reference to a task that remains in the local deque of the thread that spawned the task*

To demonstrate this, we can change how task affinities are assigned in our small example, as shown in Figure 13-5. Now, foolishly, we set all of the affinities to the same slot, the one recorded in `a[2]`.

```
void executeTaskTree(const std::string &name, bool
note_affinity,
                     bool set_affinity) {
  std::cout << name << std::endl << "id:slot" << std::endl;
  tbb::task *root = new(tbb::task::allocate_root())
tbb::empty_task;
  root->set_ref_count(numTasks+1);
  for (int i = 0; i < numTasks; ++i) {
    tbb::task *t = new ( root->allocate_child() )
                     MyTask(i,note_affinity);
    if (set_affinity) t->set_affinity(a[2]);
    tbb::task::spawn(*t);
  }
  root->wait_for_all();
}

void fig_13_5() {
  std::fill(a, a+numTasks, 0);
  executeTaskTree("note_affinity", true, false);
  executeTaskTree("with set_affinity to a[2]", false, true);
}
```

Figure 13-5. *A function that first runs different groups of tasks, sometimes noting affinities and sometimes setting affinities. An example output is also shown.*

If the TBB scheduler honors our affinity requests, there will be a large load imbalance since we have asked it to mail all of the work to the same worker thread. But if we execute this new version of the example, we see:

```
id:slot
7:0
0:1
1:2
3:4
2:3
5:5
4:6
6:7
with set_affinity to a[2]
id:slot
7:0
0:3
2:4
1:6
5:1
6:5
3:7
4:2
```

Because affinity is only a hint, the other idle threads still find tasks, stealing them from the master thread's local deque before the thread in slot `a[2]` is able to drain its affinity mailbox. In fact, only the first task spawned, `id==0`, is executed by the thread in the slot previously recorded in `a[2]`. So, we still see our tasks distributed across all eight of the threads.

The TBB library has ignored our request and instead avoided the load imbalance that would have been created by sending all of these tasks to the same thread. This weak affinity is useful in practice because it lets us communicate affinities that *should* improve performance, but it still allows the library to adjust so that we don't inadvertently create a large load imbalance.

While we can use these task interfaces directly, we see in Chapter 16 that the loop algorithms provide a simplified abstraction, `affinity_partitioner` that luckily hides us from most of these low-level details.

When and How Should We Use the TBB Affinity Features?

We should use `task_scheduler_observer` objects to create thread-to-core affinity only if we are tuning for absolute best performance on a dedicated system. Otherwise, we should let the OS do its job and schedule threads as it sees fit from its global viewpoint. If we do choose to pin threads to cores, we should carefully weigh the potential impact of taking this flexibility away from the OS, especially if our application runs in a multiprogrammed environment.

For task-to-thread affinity, we typically want to use the high-level interfaces, like `affinity_partitioner` described in Chapter 16. The `affinity_partitioner` uses the features described in this chapter to track where tasks are executed and provide hints to the TBB scheduler to replay the partitioning on subsequent executions of the loop. It also tracks changes to keep the hints up to date.

Because TBB task affinities are just scheduler hints, the potential impact of misusing these interfaces is far less – so we don't need to be as careful when we use task affinities. In fact, we should be encouraged to experiment with task affinity, especially through the higher-level interfaces, as a normal part of tuning our applications.

Summary

In this chapter, we discussed how we can create thread-to-core and task-to-thread affinity from within our TBB applications. While TBB does not provide an interface for handling the mechanics of setting thread-to-core affinity, its `class task_scheduler_observer` provides a callback mechanism that allows us to insert the necessary calls to our own OS-specific or portable libraries that assign affinities. Because the TBB work-stealing scheduler randomly assigns tasks to software threads, thread-to-core affinity is not always sufficient on its own. We therefore also discussed the interfaces in TBB's `class task` that lets us provide affinity hints to the TBB scheduler about what software thread we want a task to be scheduled onto. We noted that we will most likely not use these interfaces directly, but instead use the higher-level interfaces described in Chapters 16 and 17. For readers that are interested in learning more about these low-level interfaces though, we provided examples that showed how we can use the `note_affinity` and `set_affinity` functions to implement task-to-thread affinity for code that uses the low-level TBB tasking interface.

Like with many of the optimization features of the TBB library, affinities need to be used carefully. Using thread-to-core affinity incorrectly can degrade performance significantly by restricting the Operating System's ability to balance load. Using the task-to-thread affinity hints, being just hints that the TBB scheduler can ignore, might negatively impact performance if used unwisely, but much less so.

For More Information

- Posix set/get CPU affinity of a thread, http://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html
- SetThreadAffinityMask function, https://docs.microsoft.com/en-us/windows/desktop/api/winbase/nf-winbase-setthreadaffinitymask
- Portable Hardware Locality (hwloc), www.open-mpi.org/projects/hwloc/