CHAPTER 10

Using Tasks to Create Your Own Algorithms

One of the things that we like the most from TBB is its “multiresolution” nature. In the context of parallel programming models, multiresolution means that we can choose among different levels of abstraction to code our algorithm. In TBB, we have high-level templates, such as `parallel_for` or `pipeline` (see Chapter 2), that are ready to use when our algorithms fit into these particular patterns. But what if our algorithm is not that simple? Or what if the available high-level abstraction is not squeezing out the last drop of performance of our parallel hardware? Should we just give up and remain prisoners of the high-level features of the programing model? Of course not! There should be a capability to get closer to the hardware, a way to build our own templates from the ground up, and a way to thoroughly optimize our implementation using low-level and more tunable characteristics of the programming model. And in TBB, this capability exists. In this chapter, we will focus on one of the most powerful low-level features of TBB, the task programming interface. As we have said throughout the book, tasks are at the heart of TBB, and tasks are the building blocks used to construct the high-level templates such as `parallel_for` and `pipeline`. But there is nothing that prevents us from venturing into these deeper waters and starting to code our algorithms directly with tasks, from building our own high-level templates for future use on top of tasks, or as we describe in the next chapters, from fully optimizing our implementation by fine tuning the way in which tasks are executed. In essence, this is what you will learn by reading this chapter and the ones that follow. Enjoy the deep dive!

© Intel Corporation 2019
M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_10

A Running Example: The Sequence

Task-based TBB implementations are particularly appropriate for algorithms in which a problem can be recursively divided into smaller subproblems following a tree-like decomposition. There are plenty of problems like these. The divide-and-conquer and branch-and-bound parallel patterns (Chapter 8) are examples of classes of such algorithms. If the problem is big enough, it usually scales well on a parallel architecture because it is easy to break it into enough tasks to fully utilize hardware and avoid load unbalance.

For the purpose of this chapter, we have chosen one of the simplest problems that can be implemented following a tree-like approach. The problem is known as the Fibonacci sequence, and it consists in computing the integer sequence that starts with zero and one, and afterward, every number in the sequence is the sum of the two preceding ones:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

Mathematically, the `nth` number in the sequence, F_n, can be computed recursively as

$$F_n = F_{n-1} + F_{n-2}$$

with initial values $F_0=0$ and $F_1=1$. There are several algorithms that compute F_n, but in the interest of illustrating how TBB tasks work, we chose the one presented in Figure 10-1, although it is not the most efficient one.

```
long fib(long n) {
  if(n<2)
    return n;
  else
    return fib(n-1)+fib(n-2);
}
```

Figure 10-1. *Recursive implementation of the computation of* F_n

Fibonacci number computation is a classic computer science example for showing recursion, but it is also a classic example in which a simple algorithm is inefficient. A more efficient method would be to compute

$$F_n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

and take the upper-left element. The exponentiation over the matrix can be done quickly via repeated squaring. But, we'll go ahead in this section and use the classic recursion example for teaching purposes.

The code presented in Figure 10-1 clearly resembles the recursive equation to compute $F_n = F_{n-1} + F_{n-2}$. While it may be easy to understand, we clarify it further in Figure 10-2 where we depict the recursive calling tree when calling `fib(4)`.

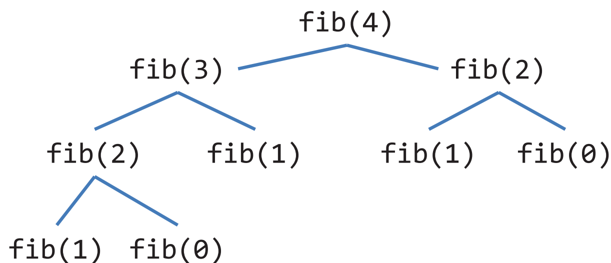


Figure 10-2. *Recursive calling tree for fib(4)*

The `if (n<2)` line at the beginning of the serial code of Figure 10-1 caters for what is called the *base case*, that is always needed in recursive codes to avoid infinite recursion, which is nice because we don't want to nuke the stack, do we?

We will parallelize this first sequential implementation using different task-based approaches, from simpler to more elaborated and optimized versions. The lessons we learn with these examples can be mimicked in other tree-like or recursive algorithms, and the optimizations we show can also be put to work to make the most out of our parallel architecture in similar situations.

The High-Level Approach: parallel_invoke

In Chapter 2, we already presented a high-level class that suits our needs when it comes to spawning parallel tasks: parallel_invoke. Relying on this class, we can come up with a first parallel implementation of the Fibonacci algorithm that we present in Figure 10-3.

```
#include <tbb/parallel_invoke.h>
long parallel_fib(long n) {
  if(n<2) {
    return n;
  }
  else {
    long x, y;
    tbb::parallel_invoke([&]{x=parallel_fib(n-1);},
                         [&]{y=parallel_fib(n-2);});
    return x+y;
  }
}
```

Figure 10-3. *Parallel implementation of Fibonacci using parallel_invoke*

The parallel_invoke member function recursively spawns parallel_fib(n-1) and parallel_fib(n-2) returning the result in stack variables x and y that are captured by reference in the two lambdas. When these two tasks finish, the caller task simply returns the sum of x+y. The recursive nature of the implementation keeps invoking parallel tasks until the base case is reached when n<2. This means that TBB will create a task even for computing parallel_fib(1) and parallel_fib(0), that just return 1 and 0 respectively. As we have said throughout the book, we want to expose enough parallelism to the architecture creating a sufficiently large number of tasks, but at the same time tasks must also have a minimum degree of granularity (>1 microsecond, as we discuss in Chapters 16 and 17) so that task creation overhead pays off. This trade-off is usually implemented in this kind of algorithm using a "cutoff" parameter as we show in Figure 10-4.

```
long parallel_fib(long n) {
  if(n<cutoff) {
    return fib(n);
  }
  else {
    long x, y;
    tbb::parallel_invoke([&]{x=parallel_fib(n-1);},
                         [&]{y=parallel_fib(n-2);});
    return x+y;
  }
}
```

Figure 10-4. *Cutoff version of the `parallel_invoke` implementation*

The idea is to modify the base case so that we stop creating more tasks when `n` is not large enough (`n<cutoff`), and in this case we resort to the serial execution. Computing a suitable `cutoff` value requires some experimentation so it is advisable to write our code so that `cutoff` can be an input parameter to ease the search of a suitable one. For example, in our test bed, `fib(30)` only takes around 1 millisecond so this is a fine-grained-enough task to discourage further splitting. Thus, it makes sense to set `cutoff=30`, which results in calling the serial version of the code for tasks that receive `n=29` and `n=28`, as we can see in Figure 10-5.

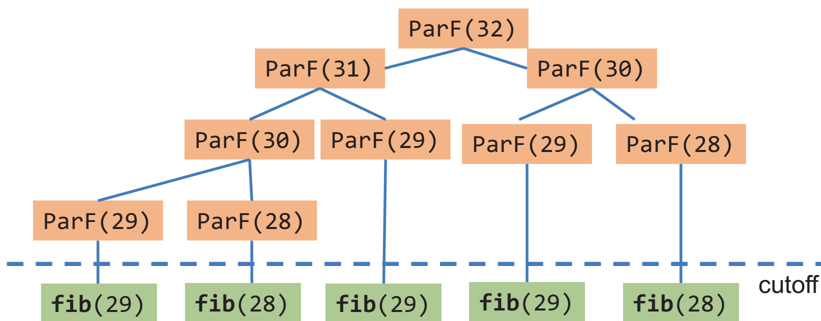


Figure 10-5. *Calling tree after invoking `parallel_fib(32)` – `ParF(32)` in the figure for the sake of saving space – `fib()` is the base case serially implemented*

If after looking at Figure 10-5 you think that it is silly to compute `fib(29)` in three different tasks and `fib(28)` in two additional ones, you are right, it is silly! As a disclaimer, we already said that this is not the optimal implementation but a commonly used recursive example that serves our educational interests. A clear optimization would

be to implement recursion in a manner such that already computed Fibonacci numbers are not recomputed again, thus achieving the optimal `O(log n)` complexity, but this is not our goal today.

You may also be thinking, after looking at Figure 10-4, why in the world we are once again revisiting the `parallel_invoke` that was already covered way back in Chapter 2. Have we *really* reached the second, more advanced, section of this book? Yes! Well… where are the advanced features, the low-level knobs that we may need and the optimization opportunities that we love??? Okay, let's start diving into deeper waters!!!

The Highest Among the Lower: `task_group`

If we can get by without some of the task knobs and optimization features that will be presented later on, the `task_group` class can serve us well. This is a higher-level and easier to use class, a medium-level abstraction, if you will. A possible re-implementation of the Fibonacci code that relies on `task_group` is presented in Figure 10-6.

```
#include <tbb/task_group.h>
long parallel_fib(long n) {
  if(n<cutoff) {
    return fib(n);
  }
  else {
    long x, y;
    tbb::task_group g;
    g.run([&]{x=parallel_fib(n-1);}); // spawn a task
    g.run([&]{y=parallel_fib(n-2);}); // spawn another task
    g.wait();                    // wait for both tasks to complete
    return x+y;
  }
}
```

Figure 10-6. *Parallel Fibonacci based on `task_group`*

Apparently, this is just a more verbose way of implementing the code of Figure 10-4 where we used `parallel_invoke`. However, we would like to underscore that, differently from the `parallel_invoke` alternative, now we have a handle to a group of tasks, g, and as we will discuss later, this enables some additional possibilities as task cancellation. Also, by explicitly calling the member functions `g.run()` and `g.wait()`, we spawn the new tasks and wait for them to finish their computation at two different program

points, whereas the `parallel_invoke` function has an implicit task barrier after the task spawning. To begin with, this separation between `run()` and `wait()` would allow for the caller thread to do some computation between spawning some tasks and waiting for them in the blocking call `wait()`. In addition, this class also offers other interesting member functions that can come handy in some situations:

- `void run_and_wait( const Func& f )`, which is equivalent to `{run(f); wait();}`, but guarantees that `f` runs on the current thread. We will see later (in section "The Low-Level Task Interface: Part Two – Task Continuation") that there is a convenient trick to bypass the TBB scheduler. If we first call `run(f)`, we basically spawn a task that gets enqueued in the worker thread local queue. When calling `wait()`, we call the scheduler that dequeues the just enqueued task if nobody else has stolen it in the meantime. The purpose of `run_and_wait` is twofold: first, we avoid the overhead of enqueueing-scheduling-dequeuing steps, and second, we avoid the potential stealing that can happen while the task is in the queue.
- `void cancel()`, which cancels all tasks in this `task_group`. Maybe the computation was triggered from a user interface, UI, that also includes a "cancel" button. If the user now presses this button, there is a way to stop the computation. In Chapter 15, we further elaborate on cancellation and exception handling.
- `task_group_status wait()`, which returns the final status of task group. Return values can be: `complete` (all tasks in the group have finished); `canceled` (`task_group` received a cancellation request); `not_completed` (not all tasks in the group have completed).

Note that, in our parallel implementation of Figure 10-6, each call to `parallel_fib` creates a new `task_group` so it is possible to cancel one branch without affecting the others, as we will see in Chapter 15. Having a single `task_group` also poses an additional downside: creating too many tasks in that single group results in task creation serialization and the ensuing loss of scalability. Consider for example we are tempted to write a code like this:

```
tbb::task_group g;
for (int i=0; i < n; i++) g.run([]{foo();});
g.wait();
```

As we see, `n` tasks will be spawn one after the other and by the same thread. The other worker threads will be forced to steal every task created by the one executing `g.run()`. This will certainly kill the performance, especially if `foo()` is a fine-grained task and the number of worker threads, `nth`, is high. The recommended alternative is the one used in Figure 10-6 where a recursive deployment of tasks is exercised. In that approach, the worker threads steal at the beginning of the computation, and ideally, in $\log_2$(`nth`) steps all `nth` worker threads are working in their own tasks that in turn enqueue more tasks in their local queues. For example, for `nth=4`, the first thread, `A`, spawns two tasks and starts working in one while thread `B` steals the other. Now, threads `A` and `B` spawn two tasks each (four in total) and start working in two of them, but the other two are stolen by threads `C` and `D`. From now on, all four threads are working and enqueueing more tasks in their local queues and stealing again only when they run out of local tasks.

BEWARE! ENTER AT YOUR OWN RISK: THE LOW-LEVEL TASKING INTERFACE

The task class has lots of features, which means there are lots of ways to make mistakes too. If the required parallel pattern is a common one, there is certainly an already available high-level template, implemented and optimized by clever developers on top of the tasking interface. This high-level algorithm is the recommended way to go in most cases. The purpose of the rest of the chapter is therefore serving two goals. In the first place, it provides you with the means to develop your own task-based parallel algorithm or high-level template if the ones already provided by TBB do not suit your needs. The second one is to uncover the low-level details of the TBB machinery so that you can understand some optimizations and tricks that will be mentioned in future chapters. For example, later chapters will refer back to this chapter to explain the way the `parallel_pipeline` and Flow Graph can better exploit locality thanks to a scheduling-bypassing technique. Here, we explain how this technique works and why it is beneficial.

The Low-Level Task Interface: Part One – Task Blocking

The TBB task class has plenty of features and knobs to fine-tune the behavior of our task-based implementation. Slowly but surely, we will introduce the different member functions that are available, progressively increasing the complexity of our Fibonacci implementation. As a starter, Figures 10-7 and 10-8 show the code required

to implement the Fibonacci algorithms using low-level tasks. This is our baseline, using task blocking style, that will be optimized in subsequent versions.

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

long parallel_fib(long n) {
  long sum;
  FibTask& a = *new(tbb::task::allocate_root()) FibTask{n,&sum};
  tbb::task::spawn_root_and_wait(a);
  return sum;
}
```

Figure 10-7. *`parallel_fib` re-implementation using the task class*

The code of Figure 10-7 involves the following distinct steps:

1. Allocate space for the task. Tasks must be allocated by special member functions so that the space can be efficiently recycled when the task completes. Allocation is done by a special overloaded `new` and `task::allocate_root` member function. The `_root` suffix in the name denotes the fact that the task created has no parent. It is the root of a task tree.
2. Construct the task with the constructor `FibTask{n,&sum}` (the task definition is presented in the next figure), invoked by `new`. When the task is run in step 3, it computes the `nth` Fibonacci number and stores it into `sum`.
3. Run the task to completion with `task::spawn_root_and_wait`.

```
class FibTask: public tbb::task {
public:
  long const n;
  long* const sum;
  FibTask(long n_, long* sum_) : n{n_}, sum{sum_} {}
  tbb::task* execute() { // Overrides virtual function task::execute
    if(n<cutoff) {
      *sum = fib(n);
    }
    else {
      long x, y;
      FibTask& a = *new(tbb::task::allocate_child()) FibTask{n-1,&x};
      FibTask& b = *new(tbb::task::allocate_child()) FibTask{n-2,&y};
      // Set ref_count to "two children plus one for the wait".
      tbb::task::set_ref_count(3);
      // Start b running.
      tbb::task::spawn(b);
      // Start a running and wait for all children (a and b).
      tbb::task::spawn_and_wait_for_all(a);
      // Do the sum
      *sum = x+y;
    }
    return nullptr;
  }
};
```

Figure 10-8. *Definition of the `FibTask` class that is used in Figure 10-7*

The real work is done inside the class `FibTask` defined in Figure 10-8. This is a relatively larger piece of code, compared to `fib` and the two previous parallel implementations of `parallel_fib`. We were advised, this is a lower-level implementation, and as such it is not as productive or friendly as a high-level abstraction. To make up for the extra burden, we will see later how this class allows us to get our hands dirty tweaking under the hood and tuning the behavior and performance at our will.

Like all tasks scheduled by TBB, `FibTask` is derived from the class `tbb::task`. The fields `n` and `sum` hold the input value and the pointer to the output, respectively. These are initialized with the arguments passed to the constructor `FibTask(long n_, long *sum_)`.

The `execute` member function does the actual computation. Every task must provide a definition of `execute` that overrides the pure virtual member function `tbb::task::execute`. The definition should do the work of the task and return either

`nullptr` or a pointer to the next task to run, as we saw in Figure 9-14. In this simple example, it returns `nullptr`.

The member function `FibTask::execute()` does the following:

1. Checks whether `n<cutoff` and resorts to the sequential version in this case.

2. Otherwise, the else branch is taken. The code creates and runs two child tasks that compute F_{n-1} and F_{n-2} respectively. Here, the inherited member function `allocate_child()` is used to allocate space for the task. Remember that the top-level routine `parallel_fib` used `allocate_root()` to allocate space for a task. The difference is that here the task is creating child tasks. This relationship is indicated by the choice of allocation method. The different allocation methods are listed in Appendix B, Figure B-76.

3. Calls `set_ref_count(3)`. The number 3 represents the two children and an additional implicit reference that is required by the member function `spawn_and_wait_for_all`. This `set_ref_count` member function initializes the `ref_count` attribute of each TBB task. Each time a child ends its computation, it decrements the `ref_count` attribute of its parent. Make sure to call `set_reference_count(k+1)` before spawning the `k` children if the task uses `wait_for_all` to be resumed after the children complete. Failure to do so results in undefined behavior. The debug version of the library usually detects and reports this type of error.

4. Spawns two child tasks. Spawning a task indicates to the scheduler that it can run the task whenever it chooses, possibly in parallel with executing other tasks. The first spawning, by the `tbb::task::spawn(b)` member function, returns immediately without waiting for the child task to start executing. The second spawning, by the member function `tbb::task::spawn_and_wait_for_all(a)`, is equivalent to `tbb::task::spawn(a); tbb::task::wait_for_all()`. The last member function causes the parent to wait until all currently allocated child tasks are finished. For this reason, we say this implementation follows what we call a task blocking style.

5. After the two child tasks complete, the `ref_count` attribute of the parent task has been decremented twice and now its value is one. This causes the parent task to resume just after the `spawn_and_wait_for_all(a)` call, so it computes `x+y` and stores it in `*sum`.

In Figure 10-9, we illustrate this task creation and execution when spawning the `root_task FibTask(8, &sum)` having set `cutoff=7`. Assuming a single thread executing all the tasks, and some simplification in the way the stack is used, in Figure 10-9 we have a streamlined representation of the computations carried out. When `parallel_fib(8)` is invoked, the variable `sum` is stored in the stack, and the root task is allocated on the heap and constructed with `FibTask(8, &sum)`. This root task is executed by a worker thread which runs the overridden `execute()` member function. Inside this member function, two stack variables `x` and `y` are declared, and two new child tasks, `a` and `b`, are allocated and enqueued in the worker thread's local deque. In the constructor of these two tasks, we pass `FibTask(7, &x)` and `FibTask(6, &y)`, which means that the variable member `sum` of the newly created tasks will point to `FibTask(8)` stack variables `x` and `y`, respectively.

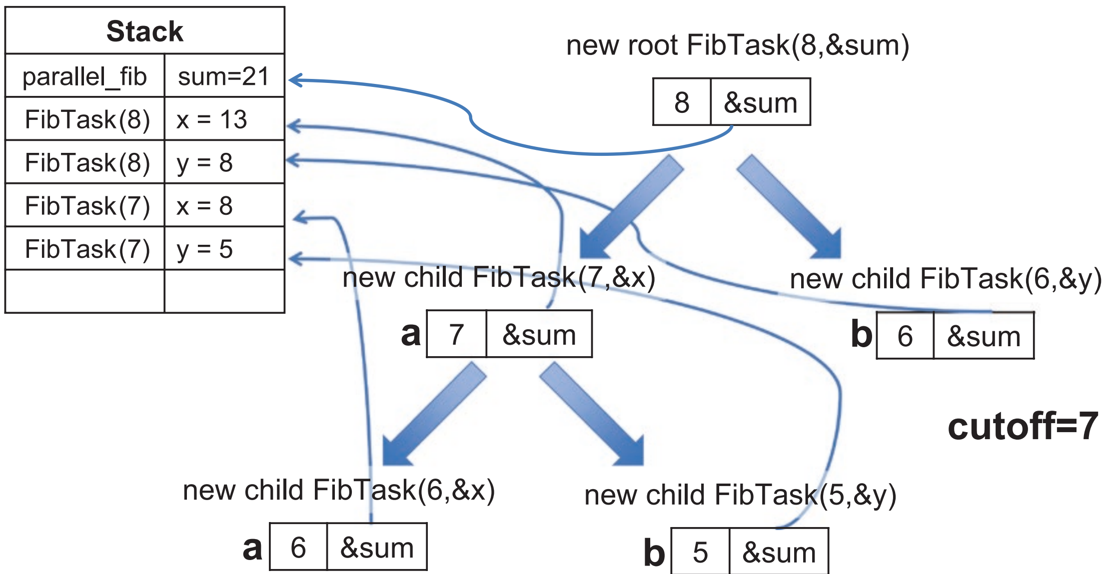


Figure 10-9. *Recursive calling tree for `parallel_fib(8)` with `cutoff=7`*

The member function `execute()` continues by setting `ref_count` of the task to `3`, spawning first `b` and then `a` and waiting for both. At this point, the root task is suspended until it has no pending child. Remember that this is the task blocking style. The worker thread returns to the scheduler, where it will first dequeue task `a` (because it was enqueued last). This task `a (FibTask(7,&x))` will recursively repeat the same process, suspending itself after allocating a new `x` and `y` on the stack and spawning `FibTask(5,&x)` and `FibTask(6,&y)`. Since `cutoff=7`, these two new tasks will resort to the base case and call `fib(5)` and `fib(6)`, respectively. `FibTask(6,&x)` is dequeued first, writes `8` to `*sum` (where `sum` points to `x` in `FibTask(7)` stack), and returns `nullptr`. Then, the `FibTask(6,&x)` is destroyed, but in the process, the `ref_cont` variable of the parent task (`FibTask(7,&x)`) is first decremented. The worker thread then dequeues `FibTask(5,&y)` that writes 5 to `*sum` (now alias of `y` in the stack) and also returns `nullptr`. This results in `ref_count` reaching the value `1`, which wakes up the parent thread `FibTask(7,&x)` that just has to add `5+8`, write it to `*sum` (alias of `x` in `FibTask(8)`) stack, and return `nullptr`. This decrements `ref_count` of the root task to 2. Next, the worker thread dequeues `FibTask(6,&y)` that calls `fib(6)`, writes `y=8` on the stack, returns, and dies. This finally leaves the root task without children (`ref_count=1`) so it can continue the execution just after the `spawn_and_wait_for_all()` member function, add `8+13`, write to `*sum` (alias of sum in the stack of `parallel_fib`), and get destroyed. If you are exhausted after reading the description of all this process, so are we, but there is even a bit more so hold on for one more second. Now, imagine that there is more than one worker thread. Each one will have its own stack and fight to steal tasks. The result, `21`, will be the same and, in essence, the same tasks will be executed, though now we don't know which thread will take care of each task. What we do know is that if the problem size and the number of tasks are large enough and if the `cutoff` is wisely set, then this parallel code will run faster than the sequential one.

Note As we have seen, the TBB work-stealing scheduler evaluates a task graph. The graph is a directed graph where each node is a task. Each task points to its parent, also called successor, which is another task that is waiting on it to complete. If a task has no parent/successor, its parent reference points to `nullptr`. Method `tbb::task::parent()` gives you read-only access to the successor pointer. Each task has a `ref_count` that accounts for the number of tasks that have it as a successor (i.e., the number of children that the parent has to wait for before it can be triggered for execution).

And where are the much-vaunted knobs and tuning possibilities? Well, it is true that the code based on low-level tasks that we just discussed is doing pretty much the same as what we already implemented with the `parallel_invoke` and `task_group` classes, but at higher programming cost. Then, where is the bang for the buck? The task class has more member functions that will be covered soon, and the implementation discussed in this section is just the foundation on which more optimized version will be built. Stick with us and keep reading.

The Low-Level Task Interface: Part Two – Task Continuation

The task blocking style that we just presented can pose a problem if the body of the task requires many local variables. These variables sit on the stack and stay there until the task is destroyed. But the task is not destroyed until all its children are done. This is a potential showstopper if the problem is very large, and it is difficult to find a cutoff value without limiting the amount of available parallelism. This can happen when facing Branch and Bound problems that are used to find an optimal value by wisely traversing a search space following a tree-based strategy. There are cases in which the tree can be very large, unbalanced (some tree branches are deeper than other), and the depth of the tree is unknown. Using the blocking style for these problems can easily result in an explosion of the number of tasks and too much use of the stack space.

Another subtle inconvenience of the blocking style is due to the management of the worker thread that encounters the `wait_for_all()` call in a parent task. There is no point in wasting this worker thread waiting for the children tasks to finish, so we entrust it with the execution of other tasks. This means that when the parent task is ready to run again, the original worker thread that was taking care of it may be distracted with other duties and cannot respond immediately.

Note *Continuation, continuation, continuation!!!* The authors of TBB, and other parallelism experts, love to encourage continuation style programming. Why??? It turns out that using it can be the difference between a working program that is relatively easy to write, and one that crashes from stack overflow. Worst yet, other than using continuations, code to solve such crashes can be hard to understand and gives parallel programming a bad name. Fortunately, TBB is designed to use

continuations and encourages us to use continuations by default. Flow Graph (Chapters 3 and 17) encourages use of `continue_node` (and other nodes with potentials for scheduler bypass). The power of continuations (and task recycling, which we cover next) is worth knowing as a parallel programmer – you'll never want to let a task sit around waiting again (and wasting precious resources)!

To avoid this problem, we can adopt a different coding style, known as continuation passing. Figure 10-10 shows the definition of a new task that we call continuation task, and Figure 10-11 underscores in boxes the required changes in `FibTask` to implement the continuation-passing style.

```
class FibCont: public tbb::task {
public:
  long* const sum;
  long x, y;
  FibCont(long* sum_) : sum{sum_} {}
  tbb::task* execute(){
    *sum = x+y;
    return nullptr;
  }
};
```

Figure 10-10. *Continuation task `FibCont` for the Fibonacci example*

The continuation task `FibCont` also has an `execute()` member function, but now it only includes the code that has to be done once the children tasks are complete. For our Fibonacci example, after the children completion, we only need to add the results that they bring and return, and these are the only two lines of code after the `spawn_and_wait_for_all(a)` in the code of Figure 10-8. The continuation task declares three member variables: a pointer to the final `sum` and the partial sum from the two children, `x` and `y`. The constructor `FibCont(long* sum)` initializes the pointer adequately. Now we have to modify our `FibTask` class to properly create and initialize the continuation task `FibCont`.

```
class FibTask: public tbb::task {
public:
  long const n;
  long* const sum;
  FibTask(long n_, long* sum_) : n{n_}, sum{sum_} {}
  tbb::task* execute() { // Overrides virtual function task::execute
    if(n<cutoff) {
      *sum = fib(n);
      return nullptr;
    }
    else {
      // long x, y; not needed anymore

      FibCont& c = *new(allocate_continuation()) FibCont{sum};
      FibTask& a = *new(c.allocate_child()) FibTask{n-1, &c.x};
      FibTask& b = *new(c.allocate_child()) FibTask{n-2, &c.y};
      // Set ref_count to "two children".
      c.set_ref_count(2);
      tbb::task::spawn(b);
      tbb::task::spawn(a);
      return nullptr;
    }
  }
};
```

Figure 10-11. *Following the continuation-passing style for parallel Fibonacci*

In Figure 10-11, besides the base case that does not change, we identify in the else part of the code that now, `x` and `y` private variables are not declared anymore and have been commented out. However, there is now a new task `c` of type `FibCont&`. This task is allocated using the `allocate_continuation()` function that is similar to `allocate_child()`, except that it transfers the parent reference of the calling task (`this`) to `c` and sets the parent attribute of `this` to `nullptr`. The reference count, `ref_count`, of `this`'s parent does not change since the parent still has the same number of children, although one of the children has suddenly been mutated from the `FibTask` type to the `FibCont` one. If you are a happy parent, don't try this at home!

At this point, `FibTask` is still alive but we will do away with it soon. `FibTask` does not have a parent anymore, but it is in charge of some chores before dying. `FibTask` first creates two `FibTask` children, but watch out!

- The new tasks `a` and `b` are now children of `c` (not of `this`) because we allocate them using `c.allocate_child()` instead of just `allocate_child()`. In other words, `c` is now the successor of both `a` and `b`.
- The result of the children is not written in stack-stored variables any more. When initializing `a`, the constructor invoked now is `FibTask(n-1,&c.x)`, so the pointer `sum` in the child task `a` (`a.sum`) is actually pointing to `c.x`. Likewise, `b.sum` points to `c.y`.
- The reference count of `c` (sort of an internal and private `c.ref_count`) is only set to two (`c.set_ref_count(2)`) since `FibCont c` has actually only two children (`a` and `b`).

Now children tasks `a` and `b` are ready to be spawned, and that's all the duties of `FibTask`. Now it can die in peace, and the memory it was occupying can be safely reclaimed. R.I.P.

Note As we mentioned in the previous section, when following the blocking style, if a task A spawns k children and waits for them using the `wait_for_all` member function, `A.ref_count` has to be set to `k+1`. The extra "`1`" accounts for the additional work that task A has to finish before ending and dispatching A's parent. This extra "`1`" is not needed when following the continuation-passing style because A transfers the additional work to the continuation task `C`. In this case, `C.ref_count` has to be set exactly to k if it has k children.

To better illustrate how all this works now that we follow the continuation-passing style, Figures 10-12 and 10-13 contain some snapshots of the process.

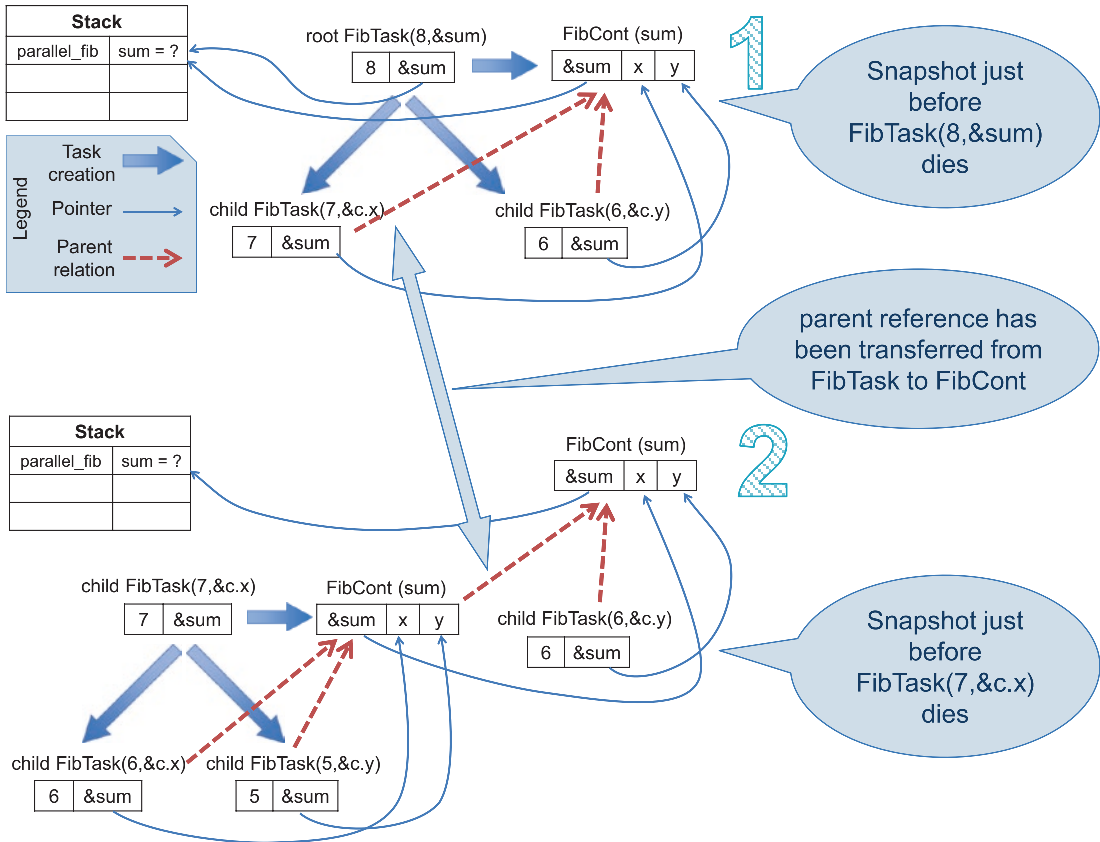


***Figure 10-12.** The continuation-passing style for `parallel_fib(8)` with `cutoff=7`*

In the upper part of Figure 10-12, the root `FibTask(8,&sum)` has already created the continuation `FibCont(sum)` and tasks `FibTask(7,&c.x)` and `FibTask(6,&c.y)`, which are actually children of `FibCont`. In the stack, we see that we are only storing the final result sum that is local to `parallel_fib` function because x and y are not using stack space using this style. Now, x and y are member variables of `FibCont` and are stored in the heap. In the bottom part of this figure, we see that the original root task has disappeared with all the memory that it was using. In essence, we are trading stack space for heap space and `FibTask`'s objects by `FibCont`'s ones, which is beneficial if `FibCont` objects are smaller. We also see that the parent reference from `FibTask(7,&c.x)` to the root `FibCont(&sum)` has been transferred to the younger `FibCont`.

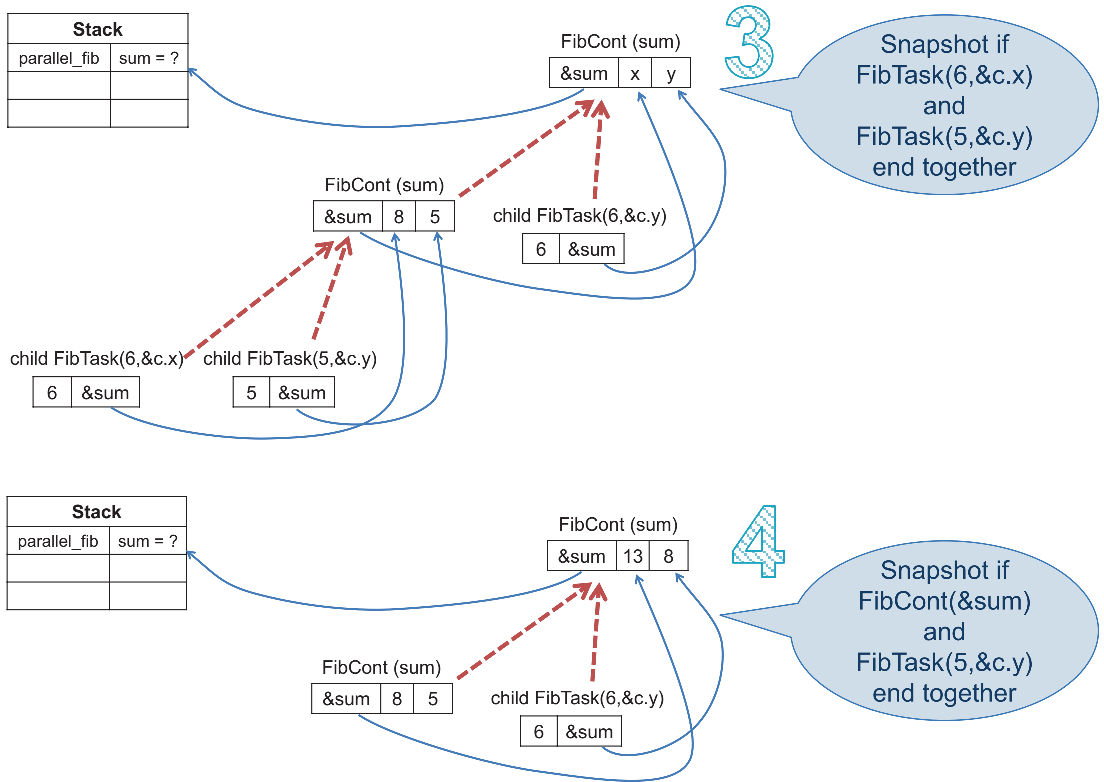


Figure 10-13. *The continuation-passing style example (continuation!!)*

In the top part of Figure 10-13, we start the unwinding part of the recursive algorithm. There is no trace of `FibTask` objects any more. Child tasks `FibTask(6,&c.x)` and `FibTask(5,&c.y)` have resorted to the base case (`n<cutoff`, assuming `cutoff=7`) and are about to return after having written `*sum` with 8 and 5, respectively. Each one of the children will return `nullptr`, so the worker thread takes control again and goes back to the work-stealing scheduler, decrements `ref_count` of the parent task, and checks whether or not `ref_count` is 0. In such a case, following the high-level description of the TBB task dispatch loop presented in Figure 9-14 of Chapter 9, the next task to take is the parent one (`FibCont` in this case). Contrary to the blocking style, this is now performed right away. In the bottom part of Figure 10-13, we see the two children of the original root task that have already written their results.

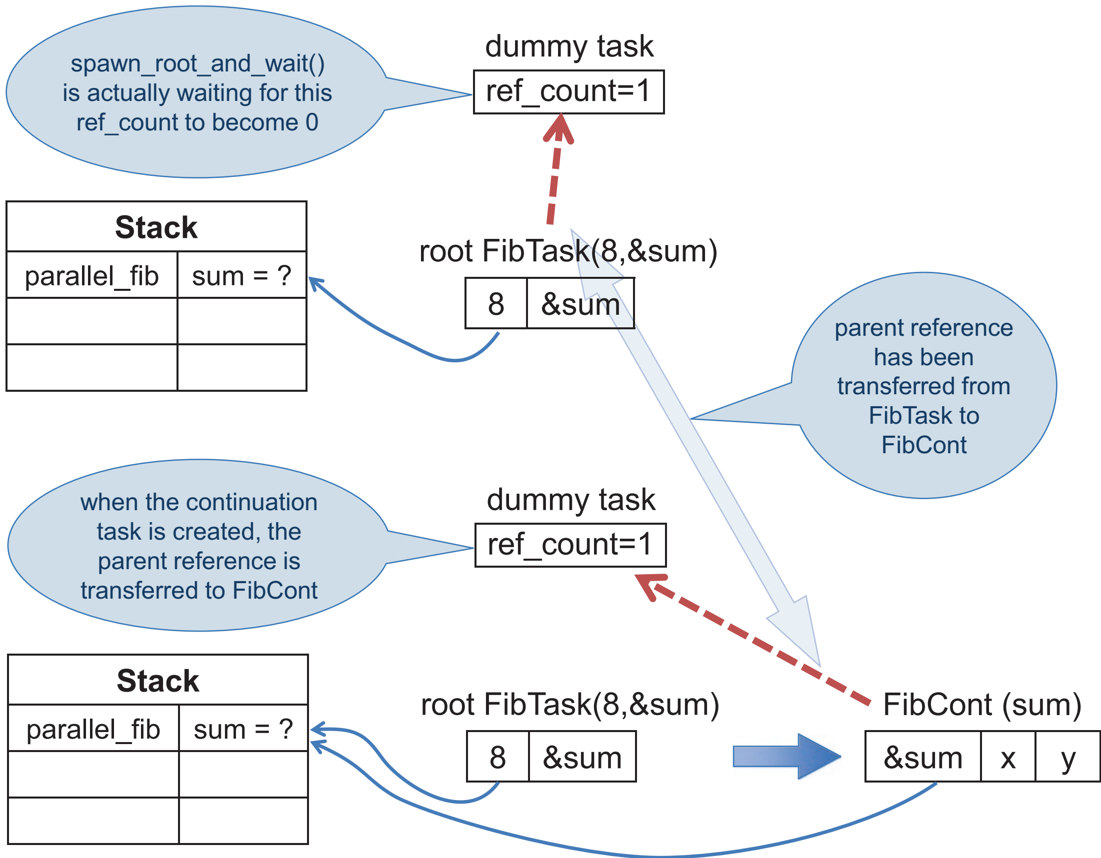


Figure 10-14. *parallel_fib waits for FibCont to complete thanks to a dummy task that has its own ref_count*

You may be wondering if the parallel_fib function is still waiting in the spawn_root_and_wait(a) to the first root task that was created, since this original FibTask was replaced by the first FibCont object and then died (see Figure 10-12). Well, indeed parallel_fib is still waiting because spawn_root_and_wait is designed to work correctly with continuation-passing style. An invocation of spawn_root_and_wait(x) does not actually wait for x to complete. Instead, it constructs a dummy successor of x and waits for the successor's ref_count to become 0. Because allocate_continuation forwards the parent reference to the continuation, the dummy successor's ref_count is not decremented until the continuation FibCont completes. This is illustrated in Figure 10-14.

Bypassing the Scheduler

Scheduler bypass is an optimization in which you directly specify the next task to run instead of letting the scheduler pick. Continuation-passing style often opens up an opportunity for scheduler bypass. For instance, in the continuation-passing example, it turns out that once `FibTask::execute()` returns, by the getting rules of the work-stealing scheduler described in Chapter 9, task `a` is always the next task taken from the ready pool because it was the last one being spawned (unless it has been stolen by another worker thread). More precisely, the sequence of events is as follows:

- Push task `a` onto the thread's deque.
- Return from member function `execute()`.
- Pop task `a` from the thread's deque, unless it is stolen by another thread.

Putting the task into the deque and then getting it back out incurs some overhead that can be avoided, or worse yet, permits stealing that can hurt locality without adding significant parallelism. To avoid both problems, make sure that `execute` does not spawn the task but instead returns a pointer to it as the result. This approach guarantees that the same worker thread immediately executes `a`, not some other thread. To that end, in the code of Figure 10-11, we need to replace these two lines as follows:

`spawn(a);`	➜	//spawn(a); commented out!
return nullptr;		return `&a;`

The Low-Level Task Interface: Part Three – Task Recycling

In addition to bypassing the scheduler, we might also want to bypass task allocation and deallocation. This opportunity frequently arises for recursive tasks that do scheduler bypass because the child is initiated immediately upon return just as the parent completes. Figure 10-15 shows the changes required to implement task recycling in the Fibonacci example.

```
class FibTask: public tbb::task {
public:
  long n;    // not const anymore
  long* sum; // not const ptr anymore
  FibTask(long n_, long* sum_) : n{n_}, sum{sum_} {}
  tbb::task* execute() { // Overrides virtual function task::execute
    if(n<cutoff) {
      *sum = fib(n);
      return nullptr;
    }
    else {
      // long x, y; not needed anymore
      FibCont& c = *new(allocate_continuation()) FibCont{sum};
      FibTask& b = *new(c.allocate_child()) FibTask{n-2, &c.y};
      recycle_as_child_of(c);
      this->n -=1;
      this->sum = &c.x;
      // Set ref_count to "two children".
      c.set_ref_count(2);
      tbb::task::spawn(b);
      return this; // it was: return &a;
    }
  }
};
```

Figure 10-15. *Following the task recycling style for parallel Fibonacci*

The child that was previously called a is now the recycled `this`. The call `recycle_as_child_of(c)` has several effects:

- It marks `this` not to be automatically destroyed when execute returns.
- It sets the successor of `this` to be `c`. To prevent reference-counting problems, `recycle_as_child_of` has a prerequisite that `this` must have a `nullptr` successor (`this`'s parent reference should point to `nullptr`). This is the case after `allocate_continuation` occurs.

Member variables have to be updated to mimic what was previously implemented using the constructor `FibTask(n-1,&c.x)`. In this case, `this->n` is decremented (`n -=1;`), and `this->sum` is initialized to point to `c.x`.

When recycling, ensure that `this`'s member variables are not used in the current execution of the task after the recycled task is spawned. This is the case in our example since the recycled task is actually not spawned and will only run after returning the pointer `this`. You can spawn the recycled task instead (i.e., `spawn (*this); return nullptr;`), as long as none of its member variables is used after the spawning. This restriction applies even to `const` member variables, because after the task is spawned, it might run and be destroyed before the parent progresses any further. A similar member function, `task::recycle_as_continuation()`, recycles a task as a continuation instead of as a child.

In Figure 10-16, we show the effect of recycling `FibTask(8,&sum)` as child of `FibCont` once the child has updated the member variables (`8` becomes `7` and sum points to `c.x`).

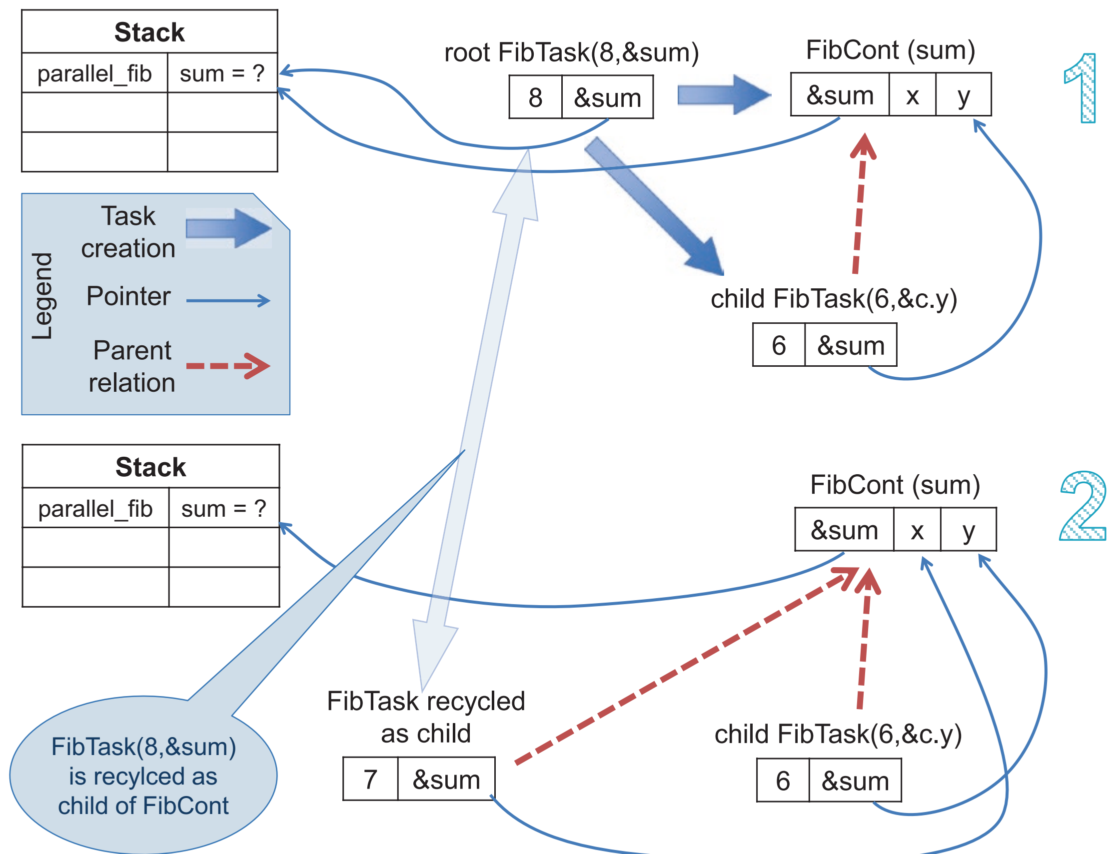


Figure 10-16. *Recycling `FibTask(8,&sum)` as a child of `FibCont`*

Note ***Greener (and easier) parallel programming*** ☺ The embracing of composability, continuations, and task recycling has a powerful impact on making parallel programming much easier simply by using TBB. Consider that recycling has gained favor around the world, and recycling of tasks really does help conserve energy too! *Join the movement for greener parallel programming – it doesn't hurt that it makes effective programming easier too!*

Scheduler bypassing and task recycling are powerful tools that can result in significant improvements and code optimizations. They are actually used to implement the high-level templates that were presented in Chapters 2 and 3, and we can also exploit them to design other tailored high-level templates that cater to our needs. Flow Graph (Chapter 3 and more coming in Chapter 17) encourages use of `continue_node` (and other nodes with potentials for scheduler bypass). In the next section, we present an example in which we leverage the low-level task API and evaluate its impact, but before that, check out our "checklist."

Task Interface Checklist

Resorting to the task interface is advisable for fork-join parallelism with lots of forks, so that the task stealing can cause sufficient breadth-first behavior to occupy threads, which then conduct themselves in a depth-first manner until they need to steal more work. In other words, the task scheduler's fundamental strategy is "breadth-first theft and depth-first work." The breadth-first theft rule raises parallelism sufficiently to keep threads busy. The depth-first work rule keeps each thread operating efficiently once it has sufficient work to do.

Remember though that it is not the simplest possible API, but one particularly designed for speed. In many cases, we face a problem that can be tackled using a higher-level interface, as the templates `parallel_for`, `parallel_reduce`, and so on do. If this is not the case and you need the extra performance offered by the task API, some of the details to remember are

- Always use `new(allocation_method) T` to allocate a task, where `allocation_method` is one of the allocation methods of class task (see Appendix B, Figure B-76). Do not create local or file-scope instances of a task.

- All siblings should be allocated before any start running, unless you are using `allocate_additional_child_of`. We will elaborate on this in the last section of the chapter.
- Exploit continuation passing, scheduler bypass, and task recycling to squeeze out maximum performance.
- If a task completes, and was not marked for re-execution (recycling), it is automatically destroyed. Also, its successor's reference count is decremented, and if it hits zero, the successor is automatically spawned.

One More Thing: FIFO (aka Fire-and-Forget) Tasks

So far, we have seen how tasks are spawned and the result of spawning a task: the thread that enqueues the task is likely the one dequeuing it in a LIFO (Last-in First-out) order (if no other thread steals the spawned task). As we said, this behavior has some beneficial implications in terms of locality and in restraining the memory footprint thanks to the "depth-first work." However, a spawned task can become buried in the local queue of the thread if a bunch of tasks are also spawned afterward.

If we prefer FIFO-like execution order, a task should be enqueued using the enqueue function instead of the spawn one, as follows:

```
class FifoTask : public tbb::task {
public:
  tbb::task *execute() {//do work}
};

FifoTask& t = *new(tbb::task::allocate_root()) FifoTask();
tbb::task::enqueue(t);
```

Our example `FifoTask` class derives from `tbb::task` and overrides the `execute()` member function as every normal task does. The four differences with spawned tasks are

- A spawned task can be postponed by the scheduler until it is waited upon, but an enqueued task will be eventually executed even if there is no thread explicitly waiting on the task. Even if the total number of worker threads is zero, a special additional worker thread is created to execute enqueued tasks.

- Spawned tasks are scheduled in a LIFO like order (most recently spawned is started next), but enqueued tasks are processed in roughly (not precisely) FIFO order (started in approximately the order they entered the queue – the "approximation" gives TBB some flexibility to be more efficient than a strict policy would allow).
- Spawned tasks are ideal for recursive parallelism in order to save memory space thanks to a depth-first traversal, but enqueued tasks can prohibitively consume memory for recursive parallelism since the recursion will expand in a breadth-first traversal.
- Spawned parent tasks should wait for their spawned children to complete, but enqueued tasks should not be waited upon because other enqueued tasks from unrelated parts of the program might have to be processed first. The recommended pattern for using an enqueued task is to have it asynchronously signal its completion. Essentially, enqueued tasks should be allocated as root, instead of as children that are then waited upon.

In Chapter 14, enqueued tasks are also illustrated in the context of prioritizing some tasks over others. Two additional use cases are also described in the Threading Building Blocks Design Patterns manual (see "For More Information" at the end of the chapter). There are two design patterns in which enqueued tasks come in handy. In the first one, a GUI thread must remain responsive even when a long running task is launched by the user. In the proposed solution, the GUI thread enqueues a task but does not wait for it to finish. The task does the heavy lifting and then notifies the GUI thread with a message before dying. The second design pattern is also related with assigning nonpreemptive priorities to different tasks.

Putting These Low-Level Features to Work

Let's switch to a more challenging application to evaluate the impact of different task-based implementation alternatives. Wavefront is a programming pattern that appears in scientific applications such as those based in dynamic programming or sequence alignment. In such a pattern, data elements are distributed on multidimensional grids representing a logical plane or space. The elements must be computed in order because they have dependencies among them. One example is the 2D wavefront that we show

in Figure 10-17. Here, computations start at a corner of the matrix, and a sweep will progress across the plane to the opposite corner following a diagonal trajectory. Each antidiagonal represents the number of computations or elements that could be executed in parallel without dependencies among them.

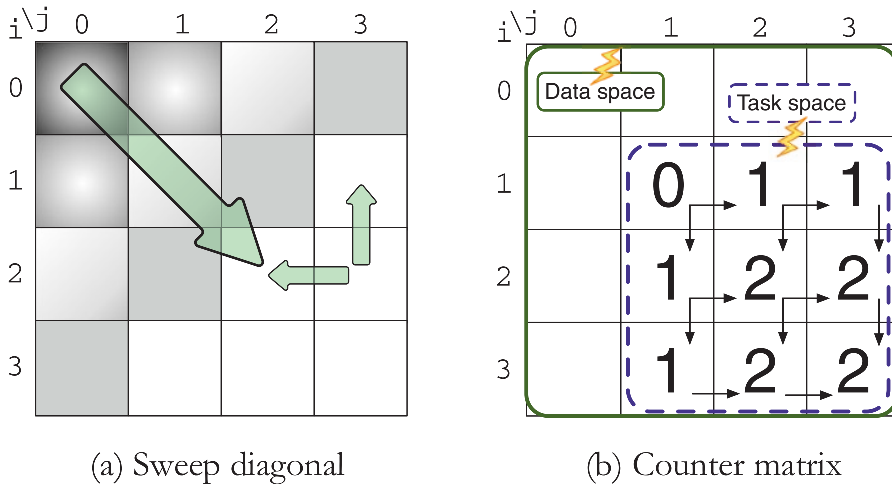


***Figure 10-17.** Typical 2D wavefront pattern (a) and dependencies translated into a matrix of atomic counters (b)*

In the code of Figure 10-18, we compute a function for each cell of a `nxn` 2D grid. Each cell has a data dependence with two elements of the adjacent cells. For example, in Figure 10-17(a), we see that cell (2,3) depends on the north (1,3) and west (2,2) ones, since on each iteration of the `i` and `j` loops, cells that were calculated in previous iterations are needed: `A[i,j]` depends on `A[i-1,j]` (north dependency) and `A[i,j-1]` (west dependency). In Figure 10-18, we show the sequential version of the computation where array A has been linearized. Clearly, the antidiagonal cells are totally independent, so they can be computed in parallel. To exploit this parallelism (loops "`i`" and "`j`"), a task will carry out the computations corresponding to each cell inside the iteration space (or task space from now on), and independent tasks will be executed in parallel.

```
    for (int i=1; i<n; ++i)
      for (int j=1; j<n; ++j)
        A[i*n+j] = foo(gs, A[i*n+j], A[(i-1)*n+j], A[i*n+j-1]);
```

Figure 10-18. *Code snippet for a 2D wavefront problem. Array A is the linearized view of the 2D grid.*

In our task parallelization strategy, the basic unit of work is the computation performed by function `foo` at each `(i,j)` cell of the matrix. Without loss of generality, we assume that the computational load for each cell will be controlled by the `gs` (grainsize) parameter of the `foo` function. That way, we can define the granularity of the tasks, and therefore, we can study the performance of different implementations depending on the task granularity, as well as situations with homogeneous or heterogeneous task workloads.

In Figure 10-17(b), the arrows show the data dependence flow for our wavefront problem. For example, after the execution of the upper left task `(1, 1)`, which does not depend on any other task, two new tasks can be dispatched (the one below `(2, 1)` and the one to the right `(1, 2)`). This dependence information can be captured by a 2D matrix with counters, like the one we show in Figure 10-17(b). The value of the counters points out to how many tasks we have to wait for. Only the tasks with the corresponding counter nullified can be dispatched.

An alternative to implement this kind of wavefront computation is covered in the Intel TBB Design Patterns (see "For More Information") in which a General Graph of Acyclic tasks is implemented. This version is available along with the sources of this chapter under the name `wavefront_v0_DAG.cpp`. However, that version requires that all the tasks are preallocated beforehand and the implementation that we present next is more flexible and can be tuned to better exploit locality as we will see later. In Figure 10-19, we show our first task-based implementation that we call `wavefront_v1_addchild`. Each ready task first executes the task body, and then it will decrement the counters of the tasks depending on it. If this decrement operation ends up with a counter equal to 0, the task is also responsible of spawning the new independent task. Note that the counters are shared and will be modified by different tasks that are running in parallel. To account for this issue, the counters are atomic variables (see Chapter 5).

```
class Cell: public tbb::task {
  int i,j;
  int n;
  int gs;
  std::vector<double> &A;
  std::vector<std::atomic<int>> &counters;
public:
  Cell(int i_ ,int j_, int n_, int gs_,
       std::vector<double> &A_,
       std::vector<std::atomic<int>> &counters_) :
       i{i_},j{j_},n{n_},gs{gs_},A{A_},counters{counters_} {}
  task* execute(){
    A[i*n+j] = foo(gs, A[i*n+j], A[(i-1)*n+j], A[i*n+j-1]);
    if (j<n-1 && --counters[i*n+j+1]==0) // east cell ready
        spawn(*new(allocate_additional_child_of(*parent()))
              Cell{i,j+1,n,gs,A,counters});
    if (i<n-1 && --counters[(i+1)*n+j]==0) // south cell ready
        spawn(*new(allocate_additional_child_of(*parent()))
              Cell{i+1,j,n,gs,A,counters});
    return nullptr;
  }
};
```

Figure 10-19. *Excerpt from the code of the wavefront_v1_addchild version*

Note that in Figure 10-19, we use `allocate_additional_child_of(*parent())` as the allocation method for the new tasks. By using this allocation method, we can add children while others are running. On the positive side, this allow us to save some coding that would have been necessary to ensure that all child tasks are allocated before any of them is spawned (since this depends on whether the east task, the south, or both are ready to be dispatched). On the negative side, this allocation method requires that the parent's `ref_count` is atomically updated (incremented when one "`additional_child`" is allocated and decremented when any child dies). Since we are using `allocate_additional_child_of(*parent())`, all created tasks will be children of the same parent. The first task of the task space is task `(1, 1)`, and it is spawned with

```
tbb::task::spawn_root_and_wait(*new(tbb::task::allocate_root())
                               Cell{1,1,n,gs,A,counters});
```

and the parent of this root task is the dummy task that we already introduced in Figure 10-14. Then, all the tasks created in this code atomically update the `ref_count` of the dummy task.

Another caveat on using the `allocate_additional_child_of` allocation method is that the user (we) has to ensure that the parent's `ref_count` does not prematurely reach `0` before the additional child is allocated. Our code already accounts for this eventuality since a task, `t`, allocating an additional child, `c`, is already guaranteeing that the `t` parent's `ref_count` is at least one since `t` will only decrement its parent's `ref_count` when dying (i.e., after allocating `c`).

In Chapter 2, the `parallel_do_feeder` template was already presented to illustrate a different wavefront application: the forward substitution. This template essentially implements a work-list algorithm, in such a way that new tasks can be added dynamically to the work-list by invoking the `parallel_do_feeder::add()` member function. We call `wavefront_v2_feeder` to a version of the wavefront code that relies on `parallel_do_feeder` and, as in Figure 2-19 in Chapter 2, uses `feeder.add()` instead of the spawn calls in Figure 10-19.

If we want to avoid all child tasks pending from a single parent and fighting to atomically update its `ref_count`, we can implement a more elaborated version that mimics the blocking style explained earlier. Figure 10-20 shows the `execute()` member function in this case, where we first annotate whether the east, south, or both cells are ready to dispatch and then allocate and dispatch the corresponding tasks. Note that now we use the `allocate_child()` method, and each task has at most two descendants to wait upon. Although the atomic update of a single `ref_count` is not a bottleneck anymore, more tasks are in flight waiting for their children to finish (and occupying memory). This version will be named wavefront_v3_blockstyle.

```
task* execute(){
  A[i*n+j] = foo(gs, A[i*n+j], A[(i-1)*n+j], A[i*n+j-1]);
  int east = 0,south = 0;
  if (j<n-1 && --counters[i*n+j+1]==0) east=1;
  if (i<n-1 && --counters[(i+1)*n+j]==0) south=1;
  set_ref_count(1+east+south);
  if(east==1 && south==0)
    spawn_and_wait_for_all(*new(allocate_child())
                           Cell{i,j+1,n,gs,A,counters});
  if(east==0 && south==1)
    spawn_and_wait_for_all(*new(allocate_child())
                           Cell{i+1,j,n,gs,A,counters});
  if(east==1 && south==1) {
    //ensure all children are allocated before any is spawned
    Cell &a = *new(allocate_child()) Cell{i,j+1,n,gs,A,counters};
    Cell &b = *new(allocate_child()) Cell{i+1,j,n,gs,A,counters};
    spawn(a);
    spawn_and_wait_for_all(b);
  }
  return nullptr;
}
```

Figure 10-20. *execute() member function of the wavefront_v3_blockstyle version*

Now, let's also exploit continuation-passing and task recycling styles. In our wavefront pattern, each task has the opportunity to spawn two new tasks (east and south neighbors). We can avoid the spawn of one of them by returning a pointer to the next task, so instead of spawning a new task, the current task recycles into the new one. As we have explained, with this we achieve two goals: reducing the number of task allocations, calls to `spawn()`, as well as saving the time for getting new tasks from the local queue. The resulting version is called `wavefront_v4_recycle`, and the main advantage is that it reduces the number of spawns from `n x n – 2n` (the number of spawns in previous versions) to `n – 2` (approximately the size of a column). See the companion source code to have a look at the complete implementation.

In addition, when recycling we can provide hints to the scheduler about how to prioritize the execution of tasks to, for example, guarantee a cache-conscious traversal of the data structure, which might help to improve data locality. In Figure 10-21, we see the code snippet of the `wavefront_v5_locality` version that features this optimization. We set the flag `recycle_into_east` if there is a ready to dispatch task to the east of the executing task. Otherwise, we set the flag `recycle_into_south`, if the south task is ready

to dispatch. Later, according to these flags, we recycle the current task into the east or south tasks. Note that, since in this example the data structure is stored by rows, if both east and south tasks are ready, the data cache can be better exploited by recycling into the east task. That way, the same thread/core executing the current task is going to take care of the task traversing the neighbor data, so we make the most out of spatial locality. So, in that case, we recycle into the east task and spawn a new south task that would be executed later.

```
task* execute(){
  A[i*n+j] = foo(gs, A[i*n+j], A[(i-1)*n+j], A[i*n+j-1]);
  bool recycle_into_east=false;
  bool recycle_into_south=false;
  if (j<n-1 && --counters[i*n+j+1]==0) recycle_into_east=true;
  if (i<n-1 && --counters[(i+1)*n+j]==0){
      if (!recycle_into_east) recycle_into_south = true;
      else
        spawn(*new(allocate_additional_child_of(*parent()))
              Cell{i+1,j,n,gs,A,counters});
    }
  if (recycle_into_east) {
    recycle_as_child_of(*parent());
    j = j+1;
    return this;
  }
  else if(recycle_into_south){
    recycle_as_child_of(*parent());
    i=i+1;
    return this;
  }
  else return nullptr;
}
```

Figure 10-21. *execute() member function of the wavefront_v5_locality version*

For huge wavefront problems, it may be relevant to reduce the footprint of each allocated task. Depending on whether or not you feel comfortable using global variables, you can consider storing the shared global state of all the tasks (`n`, `gs`, `A`, and `counters`) in global variables. This alternative is implemented in `wavefront_v6_global`, and it is provided in the directory with the source code of this chapter's examples.

Using the parameter gs that sets the number of floating-point operations per task, we found that for coarse-grained tasks that execute more than 2000 floating-point operations (FLOPs), there is not too much difference between the seven versions and the codes scale almost linearly. This is because the parallel overheads vanish in comparison with the large enough time needed to compute all the tasks. However, it is difficult to find real wavefront codes with such coarse-grained tasks. In Figure 10-22, we show the speedup achieved by versions 0 to 5 on a quad-core processor, more precisely, a Core i7-6700HQ (Skylake architecture, 6th generation) at 2.6 GHz, 6 MB L3 cache, and 16 GB RAM. Grain size, gs, is set to only 200 FLOPs and n=1024 (for this n, version 6 performs as version 5).

	Seq.	v0 DAG	v1 addchild	v2 feeder	v3 blockstyle	v4 recycle	v5 locality
Time (ms)	240	112	129	130	96	74	69
Speed-up	1	2.14	1.86	1.85	2.50	3.24	3.48

Figure 10-22. *Speedup on four cores of the different versions*

It is clear that TBB v5 is the best solution in this experiment. In fact, we measured speedups for other finer-grained sizes finding that the finer the granularity, the better the improvement of v4 and v5 in comparison with v1 and v2. Besides, it is interesting to see that a great deal of the improvement contribution is due to the recycling optimization, pointed out by the v4 enhancement over the v1 version. A more elaborated study was conducted by A. Dios in the papers that are listed at the end of the chapter.

Since the performance of the wavefront algorithms decreases as the task workload grain becomes finer, a well-known technique to counteract this trend is tiling (see the Glossary for a brief definition). By tiling, we achieve several goals: to better exploit locality since each task works within a space confined region of data for some time; to reduce the number of tasks (and therefore, the number of allocations and spawns); and to save some overhead in wavefront bookkeeping (memory space and the initialization time of the counter/dependence matrix, which is now smaller due to it requiring a counter per block-tile, and not one per matrix element). After coarsening the grain of the tasks via tiling, we are again free to go for v1 or v2 implementations, right? However,

the downside of tiling is that it reduces the amount of independent task (they are coarser, but there are fewer of them). Then, if we need to scale our application to a large number of cores and the problem does not grow in size at the same pace, we probably have to squeeze until the last drop of available performance out of the TBB low-level features. In challenging situations like these, we have to demonstrate our outstanding command of TBB and that we have successfully honed our parallel programming skills.

Summary

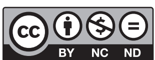
In this chapter, we have delved into the task-based alternatives that can be particularly useful to implement recursive, divide and conquer, and wavefront applications, among others. We have used the Fibonacci sequence as a running example that we first implemented in parallel with the already discussed high-level `parallel_invoke`. We then started diving into deeper waters by using a medium-level API provided by the `task_group` class. It is though the task interface the one offering the larger degree of flexibility to cater for our specific optimization needs. TBB tasks are underpinning the other high-level templates that were presented in the first part of the book, but we can also get our hands on them to build our own patterns and algorithms, leveraging continuation passing, scheduler bypassing, and task recycling advanced techniques. For even more demanding developers, more possibilities are available thanks to task priorities, task affinity, and task enqueue features that will we cover in the next chapter. We can't wait to see what you can create and develop out of these powerful tools that are now in your hands.

For More Information

Here are some additional reading materials we recommend related to this chapter:

- A. Dios, R. Asenjo, A. Navarro, F. Corbera, E.L. Zapata, A case study of the task-based parallel wavefront pattern, Advances in Parallel Computing: Applications, Tools and Techniques on the Road to Exascale Computing, ISBN: 978-1-61499-040-6, Vol. 22, pp. 65–72, IOS Press BV, Amsterdam, 2012 (extended version available here: www.ac.uma.es/~compilacion/publicaciones/UMA-DAC-11-02.pdf).

- A. Dios, R. Asenjo, A. Navarro, F. Corbera, E.L. Zapata *High-level template for the task-based parallel wavefront pattern,* IEEE Intl. Conf. on High Performance Computing (HiPC 2011), Bengaluru (Bangalore), India, December 18–21, 2011. Implement a high-level template on top of TBB task to ease the implementation of wavefront algorithms.
- González Vázquez, Carlos Hugo, Library-based solutions for algorithms with complex patterns of parallelism, PhD report, 2015. `http://hdl.handle.net/2183/14385`. Describes three complex parallel patterns and addresses them by implementing high-level templates on top of TBB tasks.
- Intel TBB Design Patterns:
    - GUI thread: `http://software.intel.com/en-us/node/506119`
    - Priorities: `http://software.intel.com/en-us/node/506120`
    - Wavefront: `http://software.intel.com/en-us/node/506110`


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.