CHAPTER 15

Cancellation and Exception Handling

More or less frequently, we all get bitten by run-time errors, either in our sequential or parallel developments. To try to assuage the pain, we have learnt to capture them using error codes or a more high-level alternative like exception handling. C++, as most OO languages, supports exception handling, which, when conveniently exercised, enables the development of robust applications. Now, considering that TBB adds task-based parallelism on top of C++, it is perfectly understandable that developers should expect that exception handling is well supported. As we will see in this chapter, exception handling is indeed well and automatically supported in TBB. This means that in case of an error, perish the thought, our code can resort to an exception handler if such is available, or terminate the whole work otherwise. Implementing support in TBB was certainly nontrivial considering that

1. Exceptions can be thrown inside of tasks that are executed by a number of threads.
2. Cancellation of tasks has to be implemented in order to terminate the work that threw the exception.
3. TBB composability has to be preserved.
4. Exception management should not affect performance if no exception arises.

The implementation of exceptions within TBB meets all these requirements, including the support of task cancellation. As we said, task cancellation support is necessary since throwing an exception can result in the need to cancel the execution of

M. Voss, R. Asenjo, J. Reinders, *Pro TBB*, https://doi.org/10.1007/978-1-4842-4398-5_15

the parallel algorithm that has generated the exception. For example, if a `parallel_for` algorithm incurs in an out-of-bound or division by zero exception, the library may need to cancel the whole `parallel_for`. This requires TBB to cancel all of the tasks involved in processing chunks of the parallel iteration space and then jump to the exception handler. TBB's implementation of task cancellation seamlessly achieves the necessary cancellation of tasks involved in the offending `parallel_for` without affecting tasks that are executing unrelated parallel work.

Task cancellation is not only a requirement for exception handling but has a value in its own. Therefore, in this chapter, we begin by showing how cancellation can be leveraged to speed up some parallel algorithms. Although cancellation of TBB algorithms just work out-of-the-box, advanced TBB developers might want to know how to get full control of task cancellation and how it is implemented in TBB. We also try to satisfy advanced developers in this chapter (remember this is the advanced part of the book). The second part of this chapter moves on to cover exception handling. Again, exception handling "just works" without any added complication: relying on our well-known try-catch construction (as we do in sequential codes) is all we need to be ready to capture standard C++ predefined exceptions plus some additional TBB ones. And again, we don't settle for the basics in this respect either. To close the chapter, we describe how to build our own custom TBB exceptions and delve into how TBB exception handling and TBB cancellation interplay under the hood.

Even if you are skeptical of exception handling because you belong to the "error code" school of thought, keep reading and discover if we end up convincing you of the advantages of TBB exception handling when developing reliable, fault-tolerant parallel applications.

How to Cancel Collective Work

There are situations in which a piece of work has to be canceled. Examples range from external reasons (the user cancels the execution by pressing a GUI button) to internal ones (an item has been found, which alleviates the need for any further searching). We have seen such situations in sequential code, but they also arise in parallel applications. For example, some expensive global optimization algorithms follow a branch-and-bound parallel pattern in which the search space is organized as a tree and we may wish to cancel the tasks traversing some branches if the solution is likely to be found in a different branch.

Let's see how we can put cancellation to work with a somewhat contrived example: we want to find the position of the single `-2` in a vector of integers, `data`. The example is contrived because we set `data[500]=-2`, so we do know the output beforehand (i.e., where –2 is stored). The implementation uses a `parallel_for` algorithm as we see in Figure 15-1.

```
std::vector<int> data(n);
data[500] = -2;
int index = -1;
auto t1 = tbb::tick_count::now();
tbb::parallel_for(tbb::blocked_range<int>{0, n},
  [&](const tbb::blocked_range<int>& r){
      for(int i=r.begin(); i!=r.end(); ++i){
        if(data[i] == -2) {
          index = i;
          tbb::task::self().cancel_group_execution();
          break;
        }
      }
});
auto t2 = tbb::tick_count::now();
std::cout << "Index "     << index;
std::cout << " found in " << (t2-t1).seconds() << " seconds!\n";
```

Figure 15-1. *Finding the index in which –2 is stored*

The idea is to cancel all other concurrent tasks collaborating in the `parallel_for` when one of them finds that `data[500]==-2`. So, what does `task::self().cancel_group_execution()`? Well, `task::self()` returns a reference to the innermost task that the calling thread is running. Tasks have been covered in several chapters, but details were provided in Chapters 10–14. In those chapters, we saw some of the member functions included in the task class, and `cancel_group_execution()` is just one more. As the name indicates, this member function does not cancel just the calling task, but **all** the tasks belonging to the same group.

In this example, the group of tasks consists of all the tasks collaborating in the `parallel_for` algorithm. By canceling this group, we are stopping all its tasks and essentially interrupting the parallel search. Picture the task that finds `data[500]==-2` shouting to the other sibling tasks "Hey guys, I got it! don't search any further!". In general, each TBB algorithm creates its own group of tasks, and every task collaborating in this TBB algorithm belongs to this group. That way, any task of the group/algorithm can cancel the whole TBB algorithm.

For a vector of size `n=1,000,000,000`, this loop consumes `0.01` seconds, and the output can be like

```
Index 500 found in 0.01368 seconds!
```

However, if `task::self().cancel_group_execution()` is commented out, the execution time goes up to `1.56` seconds on the laptop on which we happen to be writing these lines.

That's it. We are all set. That is all we need to know to do (basic) TBB algorithm cancellation. However, now that we have a clear motivation for canceling tasks (more than 100× speedup in the previous example!), we can also (optionally) dive into how task cancellation is working and some considerations to fully control which tasks actually get canceled.

Advanced Task Cancellation

In Chapter 14, the `task_group_context` concept was introduced. Every task belongs to one and only one `task_group_context` that, for brevity, we will call TGC from now on. A TGC represents a group of tasks that can be canceled or have their priority level set. In Chapter 14, some examples illustrated how to change the priority level of a TGC. We also said that a TGC object can optionally be passed to high-level algorithms like the `parallel_for` or flow graph. For instance, an alternative way to write the code of Figure 15-1 is sketched in Figure 15-2.

```
tbb::task_group_context tg;
…
tbb::parallel_for(tbb::blocked_range<int>{0, n},
  [&](const tbb::blocked_range<int>& r){
      for(int i=r.begin(); i!=r.end(); ++i){
        if(data[i] == -2){
          index = i;
          tg.cancel_group_execution();
          break;
        }
      }
}, tg);
```

Figure 15-2. *Alternative implementation of the code in Figure 15-1*

In this code, we see that a TGC, `tg`, is created and passed as the last argument of the `parallel_for`, and also used to call `tg.cancel_group_execution()` (now using a member function of the `task_group_context` class).

Note that the codes of Figures 15-1 and 15-2 are completely equivalent. The optional TGC parameter, `tg`, passed as the last argument of the `parallel_for`, just opens the door to more elaborated developments. For example, say that we also pass the same TGC variable, `tg`, to a `parallel_pipeline` that we launch in a parallel thread. Now, any task collaborating either in the `parallel_for` or in the `parallel_pipeline` can call `tg.cancel_group_execution()` to cancel both parallel algorithms.

A task can also query the TGC to which it belongs by calling the member function `group()` that returns a pointer to the TGC. That way, we can safely add this line inside the lambda of the `parallel_for` in Figure 15-2: `assert(task::self().group()==&tg);`. This means that the following three lines are completely equivalent and can be interchanged in the code of Figure 15-2:

```
tg.cancel_group_execution();
tbb::task::self().group()->cancel_group_execution();
tbb::task::self().cancel_group_execution();
```

When a task triggers the cancellation of the whole TGC, spawned tasks waiting in the queues are finalized without being run, but already running tasks will not be canceled by the TBB scheduler because, as you certainly remember, the scheduler is non-preemptive. This is, before passing the control to the `task::execute()` function, the scheduler checks the cancellation flag of the task's TGC and then decides if the task should be executed or the whole TGC canceled. But if the task already has the control, well, it has the control until it deigns to return it to the scheduler. However, in case we want to also do away with running tasks, each task can pool the canceling status using one of these two alternatives:

```
if (task::self().group()->is_group_execution_cancelled()) return;
if (task::self().is_cancelled()) return;
```

Next question: to which TGC are the new tasks assigned? Of course, we have the devices to fully control this mapping, but there is also a default behavior that is advisable to know. First, we cover how to manually map tasks into a TGC.

Explicit Assignment of TGC

As we have seen, we can create TGC objects and pass them to the high-level parallel algorithms (`parallel_for,...`) and to the low-level tasking API (`allocate_root()`). Remember that in Chapter 10 we also presented the `task_group` class as a medium-level API to easily create tasks sharing a TGC that can be canceled or assigned a priority simultaneously with a single action. All the tasks launched using the same `task_group::run()` member function will belong to the same TGC, and therefore one of the tasks in the group can cancel the whole gang.

As an example, consider the code of Figure 15-3 in which we rewrite the parallel search of a given value "hidden" in a `data` vector, and get the index in which it is stored. This time, we use a manually implemented divide-and-conquer approach using the `task_group` features (the `parallel_for` approach is actually doing something similar under the hood, even if we don't see it).

```
int grainsize = 100;
std::vector<int> data;
int myindex=-1;
tbb::task_group g;

void SerialSearch(long begin, long end){
  for(int i=begin; i<end; ++i){
    if(data[i] == -2){
      myindex = i;
      g.cancel();
      break;
    }
  }
}

void ParallelSearch(long begin, long end){
  if((end-begin) < grainsize){ //cutoof equivalent
    return SerialSearch(begin, end);
  }
  else{
    long mid = begin + (end-begin)/2;
    g.run([&]{ParallelSearch(begin, mid);}); // spawn a task
    g.run([&]{ParallelSearch(mid, end);});   // spawn another task
  }
}

int main(int argc, char** argv)
{
  int n = (argc>1) ? atoi(argv[1]) : 1000;
  data.resize(n);
  data[n/2] = -2;

  auto t0 = tbb::tick_count::now();
  SerialSearch(0, n);
  auto t1 = tbb::tick_count::now();
  ParallelSearch(0, n);
  g.wait();     // wait for all spawned tasks
  auto t2 = tbb::tick_count::now();
  double t_s = (t1 - t0).seconds();
  double t_p = (t2 - t1).seconds();

  cout << "SerialSearch:   " << myindex << " Time: " << t_s << endl;
  cout << "ParallelSearch: " << myindex << " Time: " << t_p
       << " Speedup: "       << t_s/t_p << endl;
  return 0;
}
```

Figure 15-3. *Manual implementation of the parallel search using* `task_group` *class*

For the sake of expediency, the vector, `data`, the resulting index, `myindex`, and the `task_group`, g, are global variables. This code recursively bisections the search space until a certain `grainsize` (a `cutoff` value as we saw in Chapter 10). The function `ParallelSearch(begin,end)` is the one used to accomplish this parallel partitioning. When the grainsize becomes small enough (100 iterations in our example), the `SequentialSearch(begin,end)` is invoked. If the value we were looking for, –2, is found in one of the ranges traversed inside the `SequentialSearch`, all spawned tasks are canceled using `g.cancel()`. In our laptop with four cores, and for N equal to 10 million, this is the output of our algorithm:

```
SerialSearch:   5000000 Time: 0.012667
ParallelSearch: 5000000 Time: 0.000152 Speedup: 83.3355
```

5000000 is the index of the `-2` value we have found. Looking at the speedup, we can be baffled by it running 83× faster than the sequential code. However, this is one of the situations in which we are witness to a parallel implementation having to carry out less work than the sequential counterpart: once a task finds the key, no more traversal of the vector `Data` is needed. In our run, the key is in the middle of the vector, `N/2`, and the sequential version has to get to that point, whereas the parallel version starts searching in parallel at different positions, for example, `0`, `N/4, N/2, N·3/4`, and so on.

If your mind was blown by the achieved speedup, wait and see because we can do even better. Remember that `cancel()` cannot terminate already running tasks. But again, we can query from within a running task to check if a different task in the `TGC` has canceled the execution. To achieve this using the `task_group` class, we just need to insert:

```
if(g.is_canceling()) return;
```

at the beginning of the `ParallelSearch()` function. This apparently minor mod results in these execution times:

```
SerialSearch:   5000000 Time: 0.012634
ParallelSearch: 5000000 Time: 2e-06 Speedup: 6317
```

We wish we could always get that kind of parallel speedup in a quad-core machine!!

Note Advanced and seldom needed: In addition to explicitly creating a `task_group`, setting the TGC for a TBB parallel algorithm, and setting the TCG for a root task using `allocate_root`, we can also change the TGC of any task using its member function:

```
void task::change_group(task_group_context& ctx);
```

and because we can query any task's TGC using `task::group()`, we have full control to move any task to the TGC of any other task. For example, if two tasks have access to a TGC_X variable (say you have a global `task_group_context *TGC_X`) and a first task has previously executed this:

```
TGC_X=task::self().group();
```

then a second one can execute this:

```
task::self().change_group(*TGC_X);
```

Default Assignment of TGC

Now, what happens if we do not explicitly specify the TGC? Well, the default behavior has some rules:

- A thread that creates a `task_scheduler_init` (either explicitly or implicitly by using a TBB algorithm) creates its own TGC, tagged as "**isolated**." The first task executed by this thread belongs to that TGC and subsequent child tasks inherit the same parent's TGC.
- When one of these tasks invokes a parallel algorithm without explicitly passing a TGC as optional argument (e.g., `parallel_for`, `parallel_reduce`, `parallel_do`, `pipeline`, flow graph, etc.), a new TGC, now tagged as "**bound**," is implicitly created for the new tasks that will collaborate in this nested algorithm. This TGC is therefore a child *bound* to the isolated parent TGC.
- If tasks of a parallel algorithm invoke a nested parallel algorithm, a new bound child TGC is created for this new algorithm, where the parent is now the TGC of the invoking task.

An example of a forest of TGC trees automatically built by a hypothetical TBB code is depicted in Figure 15-4.

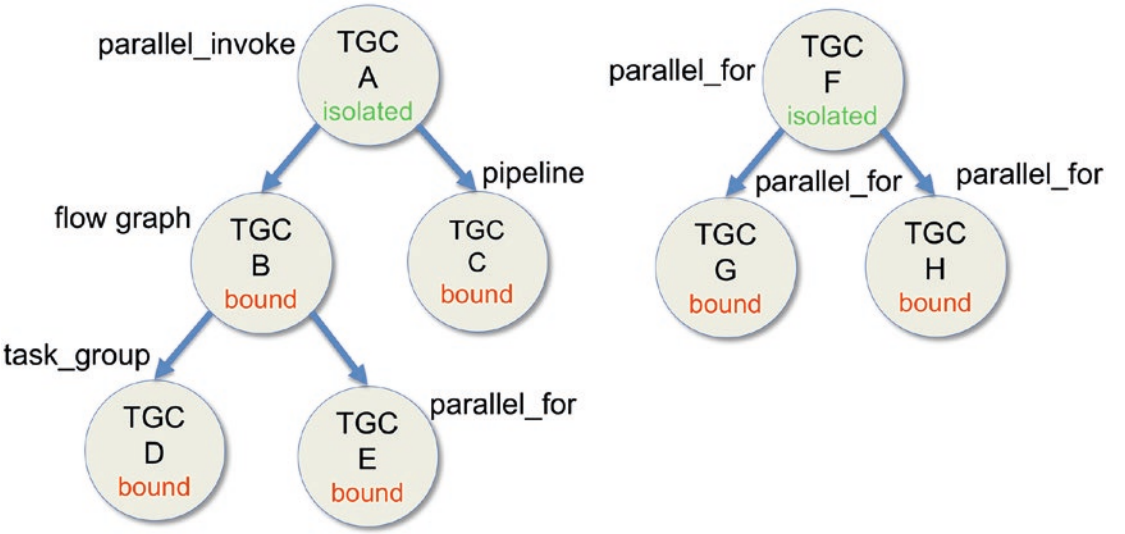

Figure 15-4. *A forest of TGC trees automatically created when running a hypothetical TBB code*

In our hypothetical TBB code, the user wants to nest several TBB algorithms but knows nothing about TGCs so he just calls the algorithms without passing the optional and explicit TGC object. In one master thread, there is a call to a `parallel_invoke`, which automatically initializes the scheduler creating one arena and the first isolated TGC, A. Then, inside the `parallel_invoke`, two TBB algorithms are created, a flow graph and a `pipeline`. For each of these algorithms, a new TGC, B and C in this case, is automatically created and bound to A. Inside one of the flow graph nodes, a `task_group` is created, and a `parallel_for` is instantiated in a different flow graph node. This results in two newly created TGCs, D and E, that are bound to B. This forms the first tree of our TGC forest, with an isolated root and where all the other TGCs are bound, that is, they have a parent. The second tree is built in a different master thread that creates a `parallel_for` with just two parallel ranges, and for each one a nested `parallel_for` is called. Again, the root of the tree is an isolated TGC, F, and the other TGCs, G and H, are bound. Note that the user just wrote the TBB code, nesting some TBB algorithms into other TBB algorithms. It is the TBB machinery creating the forest of TGCs for us. And do not forget about the tasks: there are several tasks sharing each TGC.

Now, what happens if a task gets canceled? Easy. The rule is that the whole TGC containing this task is canceled, but the cancellation also propagates downward. For example, if we cancel a task of the flow graph (`TGC B`), we will also cancel the `task_group` (`TGC D`) and the `parallel_for` (`TGC E`), as shown in Figure 15-5. It makes sense: we are canceling the flow graph, and everything created from there on. The example is somewhat contrived since it may be difficult to find a real application with this nesting of algorithms. However, it serves to illustrate how different TGCs are automatically linked in order to deal with the much vaunted TBB's composability.

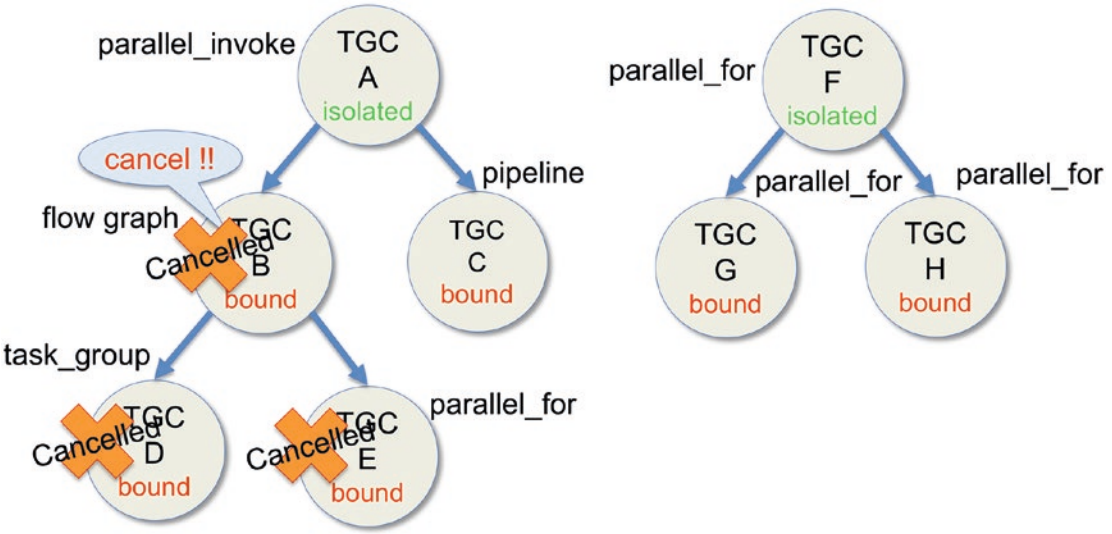

Figure 15-5. *Cancel is called from a task belonging to TGC B*

But wait, we may want to cancel the flow graph and the `task_group` but keep the `parallel_for` (`TGC E`) alive and kicking. Fine, this is also possible by manually creating an isolated TGC object and passing it as the last argument of the parallel for. To that end, we can write code similar to the one of Figure 15-6, where a `function_node` of the flow graph, g, exploits this possibility.

```
tbb::flow::function_node<float,float> node{g,…,[&](float a){
  tbb::task_group_context TGC_E(task_group_context::isolated);
  // nested parallel_for
  tbb::parallel_for(0, N, 1, [&](…){/*loop body*/}, TGC_E);
  return a;
}};
```

Figure 15-6. *Alternative to detach a nested algorithm from the tree of TGCs*

The isolated TGC object, TGC_E, is created on the stack and passed as the last argument to the parallel_for. Now, as depicted in Figure 15-7, even if a task of the flow graph cancels its TGC B, the cancellation propagates downward till TGC D but cannot reach TGC E because it has been created detached from the tree.

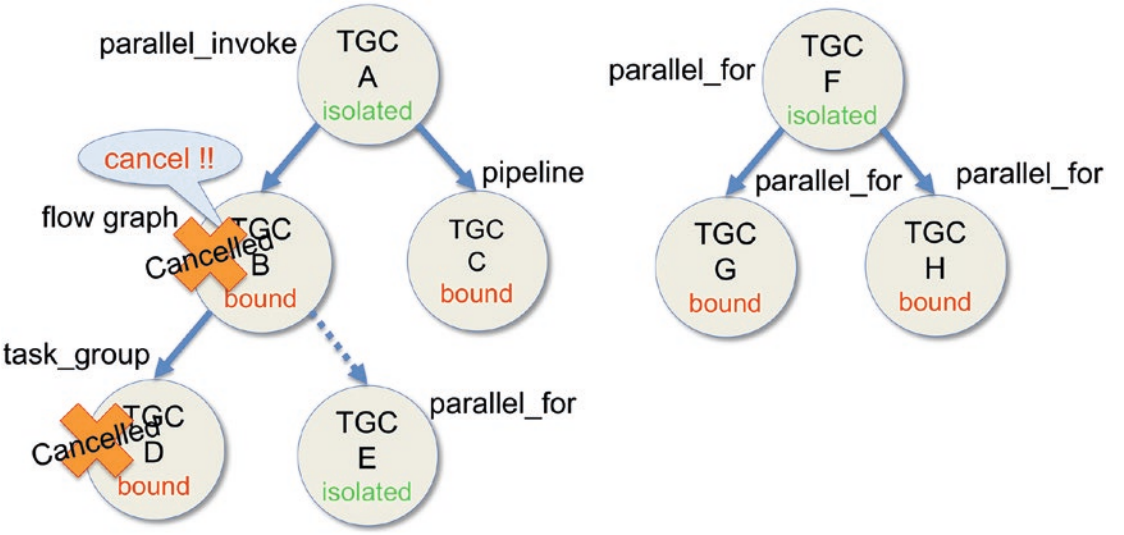

Figure 15-7. *TGC E is now isolated and won't be canceled*

More precisely, the isolated TGC E can now be the root of another tree in our forest of TGCs because it is an isolated TGC and it can be the parent of new TGCs created for deeper nested algorithms. We will see an example of this in the next section.

Summarizing, if we nest TBB algorithms without explicitly passing a TGC object to them, the default forest of TGCs will result in the expected behavior in case of cancellation. However, this behavior can be controlled at our will by creating the necessary number of TGC objects and passing them to the desired algorithms. For example, we can create a single TGC, A, and pass it to all the parallel algorithms invoked in the first thread of our hypothetical TBB example. In such a case, all tasks collaborating in all algorithms will belong to that TGC A, as depicted in Figure 15-8. If now a task of the flow graph gets canceled, not only the nested task_group and parallel_for algorithms are also canceled, but all the algorithms sharing the TGC A.

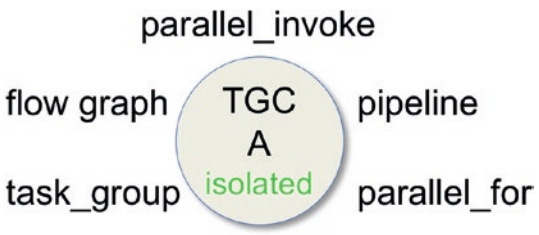

Figure 15-8. *After modifying our hypothetical TBB code so that we pass a single TGC A to all the parallel algorithms*

As a final note regarding cancellation, we want to underscore that efficiently keeping track of the forest of TGCs and how they get linked is quite challenging. The interested reader can have a look at the paper of Andrey Marochko and Alexey Kukanov (see the "For More Information" section) in which they elaborate on the implementation decisions and internal details. The main take-away is that great care was taken to ensure that TGC bookkeeping does not affect performance if cancellation is not required.

Exception Handling in TBB

Note If C++ exception is not completely familiar, here is an example to help illustrate the fundamentals:

```
int main(){
  try{
    try{
      throw 5;
    }
    catch(int& n){
      cout << "Re-throwing value: "<< n << endl;
      throw;
    }
  }
  catch(int& e){
    cout<< "Value caught: "<< e << endl;
  }
  catch (...){
    cout << "Exception occurred\n";
  }
}
```

The output after running this code is

```
Re-throwing value: 5
Value caught: 5
```

As we can see, the first try block includes a nested try catch. This one throws as an exception as an integer with value 5. Since the catch block matches the type, this code becomes the exception handler. Here, we only print the value received and re-throw the exception upward. At the outer level there are two catch blocks, but the first one is executed because the argument type matches the type of the thrown value. The second catch in the outer level receives an ellipsis (…) so it becomes the actual handler if the exception has a type not considered in the preceding chain of catch functions. For example, if we throw 5.0 instead of 5, the output message would be "Exception occurred."

Now that we understand cancellation as the keystone mechanism supporting TBB exception management, let's go into the meat of the matter. Our goal is to master the development of bulletproof code that exercise exceptions, as the one in Figure 15-9.

```
int main() {
  std::vector<int> data(1000);
  try{
    tbb::parallel_for(0, 2000, [&](int i) {data.at(i)++;});
  }
  catch(std::out_of_range& ex){
    std::cout << "Out_of_range: " << ex.what() << std::endl;
  }
  return 0;
}
```

Figure 15-9. *Basic example of TBB exception handling*

Okay, maybe it is not completely bulletproof yet, but for a first example it is good enough. The thing is that the vector `data` has only 1000 elements, but the `parallel_for` algorithm insists on walking till position 2000-1. To add insult to injury, `data` is not accessed using `data[i]`, but using `Data.at(i)`, which, contrary to the former, adds

bound-checking and throws `std::out_of_range` objects if we don't toe the line. Therefore, when we compile and run the code of Figure 15-9, we will get

```
Out_of_range: vector
```

As we know, several tasks will be spawned to increment `data` elements in parallel. Some of them will try to increment at positions beyond 999. The task that first touches an out-of-range element, for example, `data.at(1003)++`, clearly has to be canceled. Then, the `std::vector::at()` member function instead of incrementing the inexistent 1003 position throws `std::out_of_range`. Since the exception object is not caught by the task, it is re-thrown upward, getting to the TBB scheduler. Then, the scheduler catches the exception and proceeds to cancel all concurrent tasks of the corresponding TGC (we already know how the whole TGC gets canceled). In addition, a copy of the exception object is stored in the TGC data structure. When all TGC tasks are canceled, the TGC is finalized, which re-throws the exception in the thread that started the TGC execution. In our example, this is the thread that called `parallel_for`. But the `parallel_for` is in a `try` block with a `catch` function that receives an `out_of_range` object. This means that the `catch` function becomes the exception handler which finally prints the exception message. The `ex.what()` member function is responsible of returning a string with some verbose information about the exception.

Note Implementation detail. The compiler is not aware of the threading nature of a TBB parallel algorithm. This means that enclosing such algorithm in a try block results in only the calling thread (master thread) being guarded, but the worker threads will be executing tasks that can throw exceptions too. To solve this, the scheduler already includes try-catch blocks so that every worker thread is able to intercept exceptions escaping from its tasks.

The argument of the `catch()` function should be passed by reference. That way, a single catch function capturing a base class is able to capture objects of all derived types. For example, in Figure 15-9, we could have written **`catch`**`(std::exception& ex)` instead of **`catch`**`(std::out_of_range& ex)` because `std::out_of_range` is derived from `std::logic_failure` that in turn is derived from the base class `std::exception` and capturing by reference captures all related classes.

Not all C++ compilers support the exception propagation feature of C++11. More precisely, if the compiler does not support `std::exception_ptr` (as happen in a pre-C++11 compiler), TBB cannot re-throw an exact copy of the exception object. To make up for it, in such cases, TBB summarizes the exception information into a `tbb::captured_exception` object, and this is the one that can be re-thrown. There are some additional details regarding how different kinds of exceptions (`std::exception`, `tbb::tbb_exception`, or others) are summarized. However, since nowadays it is becoming difficult to get our hands on a compiler not supporting C++11, we will not pay extra attention to this TBB backward compatibility feature.

Tailoring Our Own TBB Exceptions

The TBB library already comes with some predefined exception classes that are listed in the table of Figure B-77.

However, in some cases, it is good practice to derive our own specific TBB exceptions. To this end, we could use the abstract class `tbb::tbb_exception` that we see in Figure 15-10. This abstract class is actually an interface since it declares five pure virtual functions that we are forced to define in the derived class.

```
class tbb_exception: public std::exception{
              virtual tbb_exception* move() throw() = 0;
              virtual void destroy() throw() = 0;
              virtual void throw_self() = 0;
              virtual const char* name() throw() = 0;
              virtual const char* what() throw() = 0;
          };
```

Figure 15-10. *Deriving our own exception class from `tbb::tbb_exception`*

The details of the pure virtual functions of the `tbb_exception` interface are

- `move()` should create a pointer to a copy of the exception object that can outlive the original. It is advisable to move the contents of the original, especially if it is going to be destroyed. The function specification `throw()` just after `move()` (as well as in `destroy()`, `what()`, and `name()`) is only to inform the compiler that this function won't throw anything.
- `destroy()` should destroy a copy created by `move()`.

- `throw_self()` should throw `*this`.
- `name()` typically returns the RTTI (Run-time type information) name of the originally intercepted exception. It can be obtained using the `typeid` operator and `std::type_info` class. For example, we could return `typeid(*this).name()`.
- `what()` returns a null-terminated string describing the exception.

However, instead of implementing all the virtual functions required to derive from `tbb_exception`, it is easier and recommended to build our own exception using the TBB class template, `tbb::movable_exception`. Internally, this class template implements for us the required virtual functions. The five virtual functions described before are now regular member functions that we can optionally override or not. There are however other available functions as we see in an excerpt of the signature:

```
template<typename ExceptionData>
class movable_exception: public tbb_exception{
public:
  movable_exception(const ExceptionData& src);
  ExceptionData& data() throw();
  ...
}
```

The `movable_exception` constructor and the `data()` member function will be explained with an example. Let's say that division by 0 is an exceptional event that we want to explicitly capture. In Figure 15-11, we present how we create our own exception with the help of the class template `tbb::movable_exception`.

```
class div_ex
{
  public:
    int it;
    explicit div_ex(int it_) : it{it_} {}
    const char* what() const throw(){
      return "Division by 0!";
    }
    const char* name() const throw(){
      return typeid(*this).name();
    }
};

int main(int argc, char** argv){
  int n = (argc>1) ? atoi(argv[1]): 1000;

  std::vector<float> data(n, 1.0);
  data[n/2] = 0.0;
  try {
    tbb::parallel_for(0, n, [&](int i){
      if (data[i]) data[i] = 1/data[i];
      else{
        tbb::movable_exception<div_ex> de{div_ex{i}};
        throw de;
      }
    });
  }
  catch(tbb::movable_exception<div_ex>& ex){
    std::cout << "Exception name: " << ex.data().name() << endl;
    std::cout << "Exception: "      << ex.data().what();
    std::cout << " at position: "   << ex.data().it << endl;
  }
  return 0;
}
```

Figure 15-11. *Convenient alternative to configure our own movable exception*

We create our custom class `div_ex` with the data that we want to move along with the exception. In this case, the payload is the integer `it` that will store the position at which division by 0 occurs. Now we are able to create an object, `de`, of the `movable_exception` class instantiated with the template argument `div_ex` as we do in the line:

```
tbb::movable_exception<div_ex> de{div_ex{i}};
```

where we can see that we pass a constructor of `div_ex`, `div_ex{i}`, as the argument to the constructor of `movable_exception<div_ex>`.

Later, in the catch block, we capture the exception object as `ex`, and use the `ex.data()` member function to get a reference to the `div_ex` object. That way, we have access to the member variables and member functions defined in `div_ex`, as `name()`, `what()`, and `it`. The output of this example when input parameter `n=1000000` is

```
Exception name: div_ex
Exception: Division by 0! at position: 500000
```

Although we added `what()` and `name()` as member functions of our custom `div_ex` class, now they are optional, so we can get rid of them if we don't need them. In such a case, we can change the catch block as follows:

```
catch(tbb::movable_exception<div_ex>& ex){
  cout << "Division by 0 at pos: " << ex.data().it << endl;
}
```

since this exception handler will be executed only if receiving `movable_exception<div_ex>` which only happens when a division by `0` is thrown.

Putting All Together: Composability, Cancellation, and Exception Handling

To close this chapter, let us go back to the composability aspects of TBB with a final example. In Figure 15-12, we have a code snippet showing a `parallel_for` that would traverse the rows of a matrix `Data`, were it not for the fact that it throws an exception (actually the string "oops") in the first iteration!! For each row, a nested, `parallel_for` should traverse the columns of `Data` also in parallel.

```
bool data[N][N];
int main(){
  try{
    tbb::parallel_for(0, N, 1,
      [&](int i) {
        tbb::task_group_context root{task_group_context::isolated};
        tbb::parallel_for(0, N, 1,
          [&](int j){
            data[i][j] = true;
          }
          //, root //Uncomment and see!
        );
        throw "oops";
      });
  }

  catch(...){
    std::cout << "An exception captured " << std::endl;
  }

  return 0;
}
```

Figure 15-12. *A `parallel_for` nested in an outer `parallel_for` that throws an exception*

Say that four different tasks are running four different iterations `i` of the outer loop and calling to the inner `parallel_for`. In that case, we may end up with a tree of TGCs similar to the one of Figure 15-13.

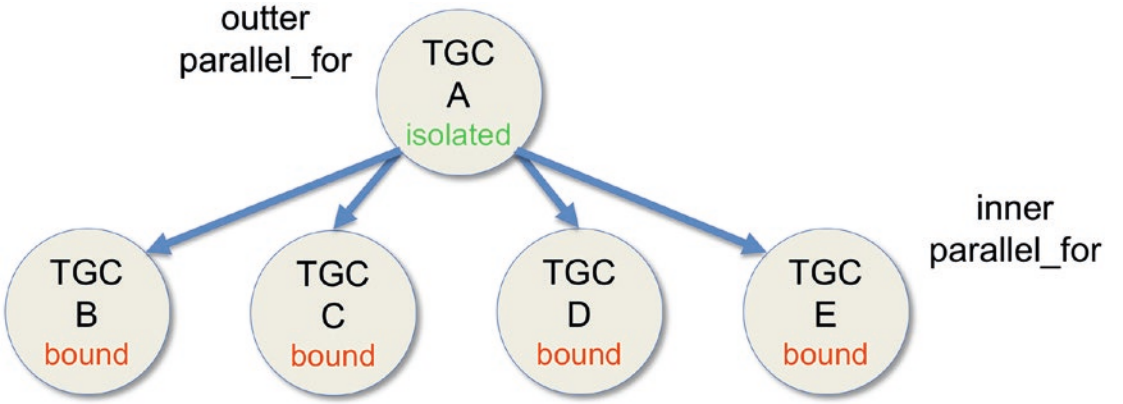

Figure 15-13. *A possible tree of TGCs for the code of Figure 15-12*

This means that when in the first iteration of the outer loop we get to the `throw` keyword, there are several inner loops in flight. However, the exception in the outer level propagates downward also canceling the inner parallel loops no matter what they are doing. The visible result of this global cancellation is that some rows that were in the process of changing the value from false to true were interrupted so these rows will have some true values and some false values.

But look, there is, per-row, an isolated `task_group_context` named `root`, thanks to this line:

```
tbb::task_group_context root(task_group_context::isolated);
```

Now, if we pass this TGC root as the last argument of the inner `parallel_for` uncommenting this line:

```
                , root //Uncomment and see!
```

We get a different configuration of the TGC, as depicted in Figure 15-14.

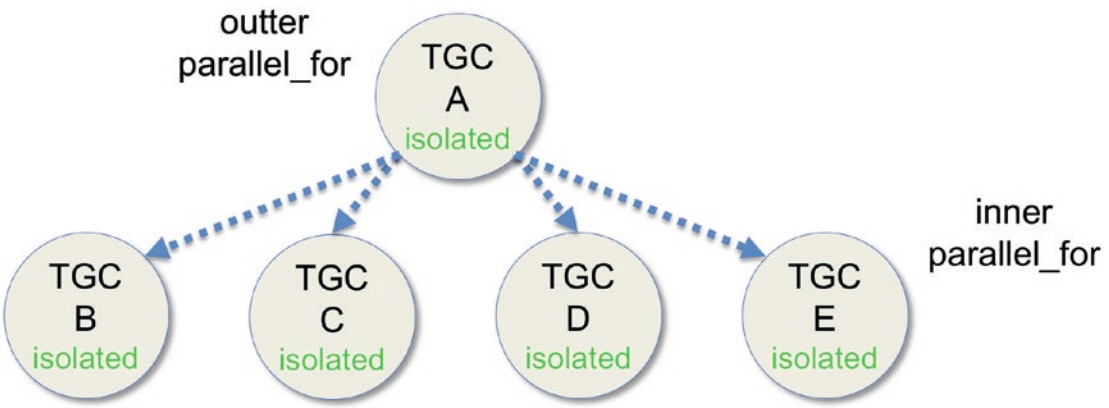

Figure 15-14. *Different configuration of the TGC*

In this new situation, the exception provokes cancellation of the TGC in which it is thrown, `TGC A`, but there are no children of `TGC A` to cancel. Now, if we check the values of the array `data` we will see that rows either have all true or all false elements, but not a mix as in the previous case. This is because once an inner loop starts setting a row with true values, it won't be canceled halfway.

In a more general case, if we can say so of our forest of TGC trees of Figure 15-4, what happens if a nested algorithm throws an exception that is not caught at any level? For example, let's suppose that in the tree of TGCs of Figure 15-15 an exception is thrown inside the flow graph (`TGC B`).

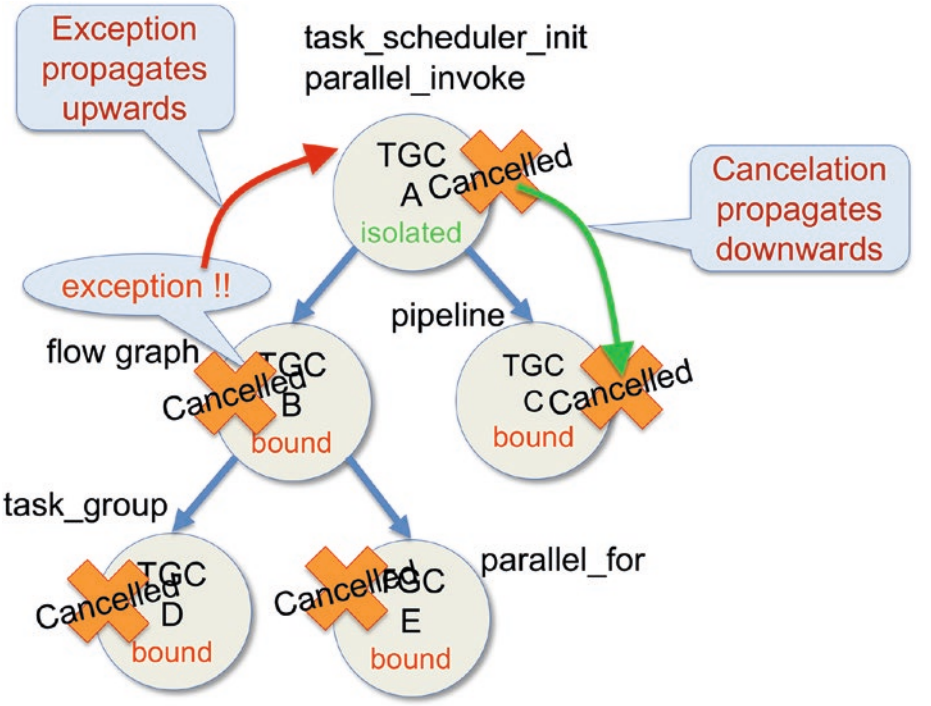

Figure 15-15. *The effect of an exception thrown in a nested TBB algorithm*

Of course, `TGC B` and descendent `TGCs D` and `E` are also canceled. We know that. But the exception propagates upward, and if at that level it is not caught either, it will provoke also the cancellation of the tasks in the `TGC A`, and because cancellation propagates downward, `TGC C` dies as well. Great! This is the expected behavior: a single exception, no matter at what level it is thrown, can gracefully do away with the whole parallel algorithm (as it would do with a serial one). We can prevent the chain of cancellations by either catching the exception at the desired level or by configuring the required nested algorithm in an isolated `TGC`. Isn't it neat?

Summary

In this chapter, we saw that canceling a TBB parallel algorithm and using exception handling to manage run-time error are straightforward. Both features just work right-out-of-the-box as expected if we resort to the default behavior. We also discussed an important feature of TBB, the task group context, TGC. This element is key in the implementation of the cancellation and exception handling in TBB and can be manually leveraged to get a closer control of these two features. We started covering the cancellation operation, explaining how a task can cancel the whole TGC to which it belongs. Then we reviewed how to manually set the TGC to which a task is mapped and the rules that apply when this mapping is not specified by the developer. The default rules result in the expected behavior: if a parallel algorithm is canceled, so are all the nested parallel algorithms. Then we moved on to exception handling. Again, the behavior of TBB exceptions resemble exceptions in sequential code, though the internal implementation in TBB is way more complex since an exception thrown in one task executed by one thread may end up being captured by a different thread. When the compiler supports C++11 features, an exact copy of the exception can be moved between threads, otherwise, a summary of the exception is captured in a `tbb::captured_exception` so that it can be re-thrown in a parallel context. We also described how to configure our own exception classes using the class template `tbb::movable_exception`. Finally, we closed the chapter by elaborating on how composability, cancellation, and exception handling interplay.

For More Information

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

- A. Marochko and A. Kukanov, Composable Parallelism Foundations in the Intel Threading Building Blocks Task Scheduler, Advances in Parallel Computing, vol 22, 2012.
- Deb Haldar, Top 15 C++ Exception handling mistakes and how to avoid them. www.acodersjourney.com/2016/08/top-15-c-exception-handling-mistakes-avoid/.