# Intel® oneAPI Threading Building Blocks Cookbook

# Contents

**Chapter 1: oneAPI Threading Building Blocks (oneTBB) Cookbook**
Developing an Application with a Parallel Execution ..... 3

# oneAPI Threading Building Blocks (oneTBB) Cookbook

1

This Cookbook introduces learning materials and use-case recipes to to parallelize and scale the code with oneTBB, a flexible C++ library that simplifies the work of adding parallelism to complex applications. Refer to other documentation to learn more about oneTBB:

- oneTBB Get Started Guide
- oneTBB Developer Guide and API Reference
- Developing an Application with a Parallel Execution

## Developing an Application with a Parallel Execution

This recipe covers how you can develop an application with parallel execution. For this purpose, we calculate the sum of array elements using oneTBB. Content experts: Alexei Katranov and Pavel Kumbrasev.

### Ingredients

Here are the minimum hardware and software requirements for this recipe. Refer to oneTBB System Requirements to see all supported options.

- **Compiler**: Intel(R) oneAPI DPC++/C++ Compiler that is available with Intel oneAPI toolkits.
- **Library**: oneTBB only.
- **Operating system:**
  - Windows* 10 OS.
  - Linux* OS.
  - macOS* 10.15, 11.x.
  - Android* 9
- **Hardware:**
  - Intel(R) Core™ processor family
  - VIntel(R) Xeon® processor family

### Directions

#### Understanding the problem

If you want to parallelize the simple task of calculation of the sum of array elements, you can use oneTBB. Firstly, a serial solution to this problem may look like this:

```
int summarize(const std::vector<int>& vec) {
    int sum = 0;
    for (int i = 0; i < vec.size(); ++i) {
    sum += vec[i];
    }
    return sum;
}
```

To execute the algorithm in parallel, you need to divide it into independent parts that can be processed autonomously of each other. The simplest approach is to divide the processed elements into several parts and process each part in its stream.

However, this code has a complexity that does not allow you to do so. All elements are summed into one variable, access to which leads to a data race because one of the threads can be writing this variable simultaneously with another thread reading or writing to the same variable as well.

> **Tip** A data race is a simultaneous and unsynchronized access to the same memory location by two or more threads when one of the accesses is a modification.

An assignment operator (operator =+) consists of three operations: read from the memory, addition operation, and store result to the memory. These operations might be executed in parallel by different threads that may lead to unexpected results. The following picture shows the possible order of operations on the timeline of two threads. The main complexity is that both threads might not get the results of operations by another thread and overwrite an invalid value. C++ considers such situations as a data race, and the program behavior is undefined there. For example, as a result of the program execution, you can get four, expecting six. It means that a program can demonstrate different results depending on the sequence of operations execution.

There are many C++ interfaces to deal with data race, consider the simplest of them, a mutex. A mutex has two main interfaces: lock and unlock. Lock takes the mutex into exclusive possession, and unlock releases it, making it available to other threads. The thread that cannot take the mutex is blocked waiting for another thread to release it.

> **Tip** The mutex concept is much simpler than the atomic operations are. It allows you to create a critical section that can be executed with only one thread at any given moment. Besides, there are advanced mutexes, like shared_lock, that can in some cases increase efficiency when working with critical sections.

The code protected by a mutex is also called a critical section. An important observation is that the second thread that failed to lock the mutex does nothing important while the first one is in the critical section. Thus, the size of the critical section can greatly affect the program performance or even the overall system performance.

**Parallelizing an Application with the <thread> Library**

Try to make the last example a parallel one. To create threads, use the <thread> library from the C++ standard library.

```
int summarize(const std::vector<int>& vec) {
int num_threads = 2;
std::vector<std::thread> threads(num_threads);

int sum = 0;
std::mutex m;
auto thread_func = [&sum, &vec, &m] (int thread_id) {
    // Split origin range into two pieces
    int start_index = vec.size() / 2 * thread_id;
    int end_index = vec.size() / 2 * (thread_id + 1);
    for (int i = start_index; i < end_index; ++i) {
        // Use lock_guard that implements RAII idiom:
        // - mutex is acquired on construction (i.e. mutex.lock() is called)
        // - mutex is released by destruction (i.e. mutex.unlock() is called)
        std::lock_guard<std::mutex> lock(m);
        sum += vec[i];
     }
};

 for (int thread_id = 0; thread_id < num_threads; ++ thread_id) {
```

```
    // Start thread with a start function `thread_func` and a function argument ` thread_id`
        threads[thread_id] = std::thread(thread_func, thread_id);
    }

    std::cout << sum << std::endl;

    for (int thread_id = 0; thread_id < num_threads; ++ thread_id) {
        // Need wait for all the threads before destruction
        threads[thread_id].join();
    }
    return sum;
}
```

If you compile the program, probably, you will see an incorrect result. The reason is that the mutex, protects the code from the data race when calculating the sum, but the main thread can read the sum variable while other threads modify it. Even if you protect the reading with the mutex, it will lead to another nonobvious complexity called a race condition.

> **Tip** Race conditions are a more general term describing a situation where the result of program execution depends on the sequence of operations performed by different threads, which may change when the application is executed again.

In this case, you do not wait for the calculations to be fully completed. To resolve this problem, wait for the completion of the threads before reading the results. However, the mutex is not required for reading the total amount because the synchronization of the calculations is performed while waiting for threads (using the join function).

```
int summarize(const std::vector<int>& vec) {
int num_threads = 2;
std::vector<std::thread> threads(num_threads);

int sum = 0;
std::mutex m;
auto thread_func = [&sum, &vec, &m] (int thread_id) {
    // Split origin range into two pieces
    int start_index = vec.size() / 2 * thread_id;
    int end_index = vec.size() / 2 * (thread_id + 1);
    for (int i = start_index; i < end_index; ++i) {
        // Use lock_guard that implements RAII idiom:
        // - mutex is acquired on construction (i.e. mutex.lock() is called)
        // - mutex is released by destruction (i.e. mutex.unlock() is called)
        std::lock_guard<std::mutex> lock(m);
        sum += vec[i];
     }
};

 for (int thread_id = 0; thread_id < num_threads; ++ thread_id) {
     // Start thread with start function `thread_func` and function argument ` thread_id`
     threads[thread_id] = std::thread(thread_func, thread_id);
 }

 for (int thread_id = 0; thread_id < num_threads; ++ thread_id) {
     // Need wait for all the threads before destruction
     threads[thread_id].join();
 }
```

```
 std::cout << sum << std::endl;

 return sum;

}
```

This approach to parallelization works slower than the serial version, because for each `sum += vec[i]` you take the mutex `std:: lock_guard<std::mutex> lock (m)`. Thus, you completely serialize the calculations, that is, only one thread to run at any given time. To avoid this, you can first perform local summation within each thread, and at the end of the calculations add the result to the global sum.

```
int sum = 0;
std::mutex m;
auto thread_func = [&sum, &vec, &m] (int thread_id) {
    // Split origin range into two pieces
    int start_index = vec.size() / 2 * thread_id;
    int end_index = vec.size() / 2 * (thread_id + 1);
    int local_sum = 0;
    for (int i = start_index; i < end_index; ++i) {
        local_sum += vec[i];
    }

    // Use lock_guard that implements RAII idiom:
    // - mutex is acquired on construction (i.e. mutex.lock() is called)
    // - mutex is released by destruction (i.e. mutex.unlock() is called)
    std::lock_guard<std::mutex> lock(m);
    sum += local_sum;
};
```

**Parallelizing an Application with oneTBB**

This simple example demonstrates that parallel programming leads to several issues that cannot be observed in a serial program. Moreover, these issues are not always easily detectible or obvious. Libraries such as oneTBB simplify parallel programming in many aspects. For instance, the example can be rewritten with parallel_reduce that does not require any special synchronizations and mechanisms to avoid race conditions:

```
int summarize(const std::vector<int>& vec) {
    int sum = tbb::parallel_reduce(tbb::blocked_range<std::size_t>{0, vec.size()}, 0,
    [&vec] (const auto& r, int init) {
        for (auto i = r.begin(); i != r.end(); ++i) {
            init += vec[i];
        }
        return init;
    },
    std::plus<int>{});

    return sum;
}
```

## Conclusion

Although this example is relatively small, it shows a set of powerful simplifications that oneTBB provides. For example, oneTBB manages a thread pool that is reused between multiple invocations of parallel algorithms. In addition, parallel_reduce implements all the necessary synchronizations, and you simply needs to describe the required operation, such as std::plus<int>. oneTBB presents a set of parallel algorithms applicable to a wide range of applications. This library uses the work stealing approach to distribute tasks between threads. The main advantage of the oneTBB approach is that it makes it easy to create parallelism in various independent components of the application.