14
Parallel Algorithms

The previous chapters have focused on how to introduce concurrency and asynchrony in our programs by using threads and coroutines. This chapter focuses on parallel execution of independent tasks, which is related to but distinct from concurrency.

In earlier chapters, I stressed that I prefer standard library algorithms over handcrafted `for`-loops. In this chapter, you will see some great advantages of using standard library algorithms with the execution policies introduced with C++17.

This chapter is not going to go in depth into theories of parallelizing algorithms or parallel programming in general, as these subjects are far too complex to cover in a single chapter. Also, there are a multitude of books on this subject. Instead, this chapter is going to take a more practical approach and demonstrate how to extend a current C++ code base to utilize parallelism while preserving the readability of the code base. In other words, we do not want the parallelism to get in the way of readability; rather, we want the parallelism to be abstracted away so that parallelizing the code is only a matter of changing a parameter to an algorithm.

In this chapter, you will learn:

- Various techniques for implementing parallel algorithms

- How to evaluate the performance of parallel algorithms
- How to adapt a code base to use the parallel extensions of the standard library algorithms

Parallel programming is a complicated topic, so before starting, you need to understand the motivation for introducing parallelism in the first place.

The importance of parallelism

From a programmer's perspective, it would be very convenient if the computer hardware of today was a 100 GHz single core CPU rather than a 3 GHz multi-core CPU; we wouldn't need to care about parallelism. Unfortunately, making single-core CPUs faster and faster has hit a physical limit. So, as the evolution of computer hardware is going in the direction of multi-core CPUs and programmable GPUs, programmers have to use efficient parallel patterns in order to make the most of the hardware.

Parallel algorithms allow us to optimize our programs by executing multiple individual tasks or subtasks at the exact same time on a multi-core CPU or GPU.

Parallel algorithms

As mentioned in *Chapter 11*, *Concurrency*, the terms *concurrency* and *parallelism* can be a little hard to distinguish from each other. As a reminder, a program is said to run concurrently if it has multiple individual control flows running during overlapping time periods. On the other hand, a parallel program executes multiple tasks or subtasks simultaneously (at the exact same time), which requires hardware with multiple cores. We use parallel algorithms to optimize latency or throughput. It makes no sense to parallelize algorithms if we don't have hardware that can execute multiple tasks simultaneously to achieve better

performance. A few simple formulas will now follow to help you understand what factors need to be considered when evaluating parallel algorithms.

Evaluating parallel algorithms

In this chapter, **speedup** is defined as the ratio between a sequential and a parallel version of an algorithm, as follows:

$$Speedup = \frac{T_1}{T_n}$$

T_1 is the time it takes to solve a problem using a sequential algorithm executing at one core, and T_n is the time it takes to solve the same problem using *n* cores. *Time* refers to wall-clock time (not CPU time).

A parallel algorithm is usually more complicated and requires more computational resources (CPU time, for example) compared to its sequential equivalent. The benefits of the parallel version come from the ability to spread the algorithm onto several processing units.

With that in mind, it's also notable that not all algorithms gain the same performance boost when run in parallel. The **efficiency** of a parallel algorithm can be computed by the following formula:

$$Efficiency = \frac{T_1}{T_n * n}$$

In this formula, *n* is the number of cores executing the algorithm. Since T_1/T_n denote the speedup, the efficiency can also be expressed as *Speedup*/*n*.

If the efficiency is *1.0*, the algorithm parallelizes perfectly. For example, it means that we achieve an 8x speedup when executing a parallel algorithm on a computer with eight cores. In practice, though, there are a multitude of parameters that limit parallel execution, such as creating threads, memory bandwidth, and context switches, as mentioned in *Chapter 11*, *Concurrency.* So, typically, the efficiency is well below 1.0.

The efficiency of a parallel algorithm depends on how independently each chunk of work can be processed. For example, `std::transform()` is trivial to parallelize in the sense that each element is processed completely independently of every other. This will be demonstrated later in this chapter.

The efficiency also depends on the problem size and the number of cores. For example, a parallel algorithm may perform very poorly on small data sets due to the overhead incurred by the added complexity of a parallel algorithm. Likewise, executing a program on a great many cores might hit other bottlenecks in the computer such as memory bandwidth. We say that a parallel algorithm scales if the efficiency stays constant when we change the number of cores and/or the size of the input.

It's also important to keep in mind that not all parts of a program can be parallelized. This fact limits the theoretical maximum speedup of a program even if we had an unlimited number of cores. We can compute the maximum possible speedup by using **Amdahl's law**, which was introduced in *Chapter 3*, *Analyzing and Measuring Performance*.

Amdahl's law revisited

Here, we will apply Amdahl's law to parallel programs. It works like this: the total running time of a program can be split into two distinct parts or *fractions*:

- F_{seq} is the fraction of the program that can only be executed *sequentially*
- F_{par} is the fraction of the program that can be executed in *parallel*

Since these two fractions together make up the entire program, it means that $F_{seq} = 1 - F_{par}$. Now, Amdahl's law tells us that the **maximum speedup** of a program executing on *n* cores is:

$$Maximum\ speedup = \frac{1}{\frac{F_{par}}{n} + F_{seq}} = \frac{1}{\frac{F_{par}}{n} + (1 - F_{par})}$$

To visualize the effect of this law, the following image shows the execution time of a program with the sequential fraction at the bottom and the parallel fraction on the top. Increasing the number of cores only affects the parallel fraction, which sets a limit on the maximum speedup:

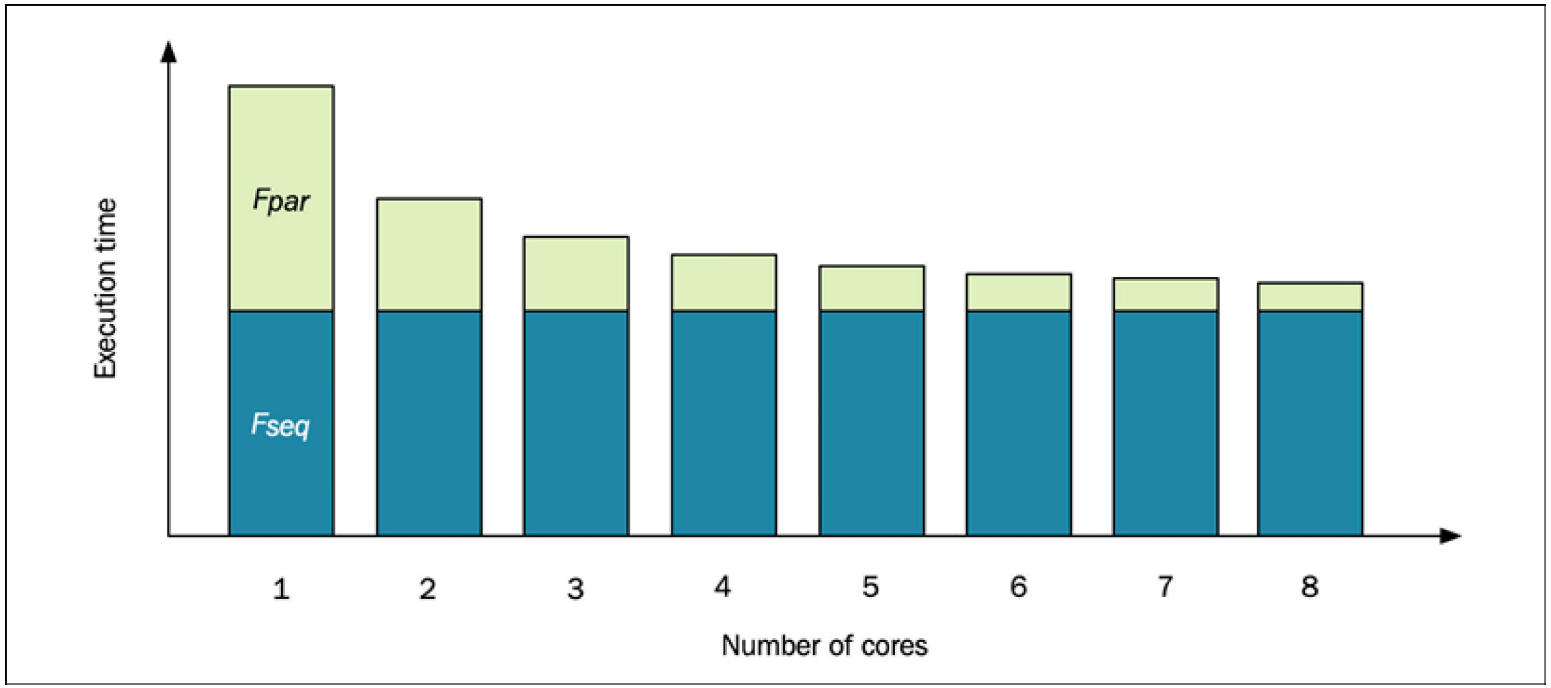

Figure 14.1: Amdahl's law defines the maximum speedup; in this case it is 2x

In the figure above, the sequential part accounts for 50% of the execution time when running on a single CPU. Therefore, the maximum speedup we can achieve by adding more cores when executing such a program is 2x.

To give you some idea of how parallel algorithms can be implemented, we will now go through a few examples. We will begin with `std::transform()` because it is relatively easy to split into multiple independent parts.

Implementing parallel std::transform()

Although algorithmically `std::transform()` is easy to implement, in practice, implementing even a rudimentary parallel version is more complex than it might appear at first sight.

The algorithm `std::transform()` calls a function for each element in a sequence, and stores the result in another sequence. A possible implementation of a sequential version of `std::transform()` may look something like this:

```
template<class SrcIt, class DstIt, class Func>
auto transform(SrcIt first, SrcIt last, DstIt dst, Func func) {
  while (first != last) {
     *dst++ = func(*first++);
  }
}
```

The standard library version also returns the `dst` iterator, but we will ignore that in our examples. To understand the challenges with a parallel version of `std::transform()`, let's begin with a naive approach.

Naive implementation

A naive parallel implementation of `std::transform()` would probably look something like this:

- Divide the elements into chunks corresponding to the number of cores in the computer
- Process each chunk in a separate task
- Wait for all tasks to finish

Using `std::thread::hardware_concurrency()` to determine the number of supported hardware threads, a possible implementation could look like this:

```
template <typename SrcIt, typename DstIt, typename Func>
auto par_transform_naive(SrcIt first, SrcIt last, DstIt dst, Func f) {
  auto n = static_cast<size_t>(std::distance(first, last));
  auto n_cores = size_t{std::thread::hardware_concurrency()};
  auto n_tasks = std::max(n_cores, size_t{1});
  auto chunk_sz = (n + n_tasks - 1) / n_tasks;
  auto futures = std::vector<std::future<void>>{};
  // Process each chunk on a separate
  for (auto i = 0ul; i < n_tasks; ++i) {
    auto start = chunk_sz * i;
    if (start < n) {
      auto stop = std::min(chunk_sz * (i + 1), n);
      auto fut = std::async(std::launch::async,
        [first, dst, start, stop, f]() {
          std::transform(first + start, first + stop, dst + start, f);
      });
      futures.emplace_back(std::move(fut));
    }
  }
  // Wait for each task to finish
  for (auto&& fut : futures) {
    fut.wait();
  }
}
```

Note that `hardware_concurrency()` might return `0` if it, for some reason, is undetermined, and therefore is clamped to be at least one.

A subtle difference between `std::transform()` and our parallel version is that they put different requirements on the iterators. `std::transform()` can operate on input and output iterators such as `std::istream_iterator<>` bound to `std::cin`. This is not possible with `par_transform_naive()` since the iterators are copied and used from multiple tasks. As you will see, there are no parallel algorithms presented in this chapter that can operate on input and output iterators. Instead, the parallel algorithms at least require forward iterators that allow multi-pass traversal.

Performance evaluation

Continuing the naive implementation, let's measure its performance with a simple performance evaluation compared to the sequential version of `std::transform()` executing at a single CPU core.

In this test we will measure the time (clock on the wall) and the total time spent on the CPUs when varying the input size of the data.

We will set up this benchmark using Google Benchmark, which was introduced in *Chapter 3*, *Analyzing and Measuring Performance*. To avoid duplicating code, we'll implement a function that will set up a test fixture for our benchmark. The fixture needs a source range with some example values, a destination range for the result, and a transform function:

```
auto setup_fixture(int n) {
  auto src = std::vector<float>(n);
  std::iota(src.begin(), src.end(), 1.0f); // Values from 1.0 to n
  auto dst = std::vector<float>(src.size());
  auto transform_function = [](float v) {
    auto sum = v;
    for (auto i = 0; i < 500; ++i) {
```

```
      sum += (i * i * i * sum);
    }
    return sum;
  };
  return std::tuple{src, dst, transform_function};
}
```

Now we have our fixture set up, it's time to implement the actual benchmark. There will be two versions: one for the sequential `std::transform()` and one for our parallel version, `par_transform_naive()`:

```
void bm_sequential(benchmark::State& state) {
  auto [src, dst, f] = setup_fixture(state.range(0));
  for (auto _ : state) {
    std::transform(src.begin(), src.end(), dst.begin(), f);
  }
}
void bm_parallel(benchmark::State& state) {
  auto [src, dst, f] = setup_fixture(state.range(0));
  for (auto _ : state) {
    par_transform_naive(src.begin(), src.end(), dst.begin(), f);
  }
}
```

Only the code within the `for`-loops will be measured. By using `state.range(0)` for input size, we can generate different values by appending a range of values to each benchmark. In fact, we need to specify a couple of arguments for each benchmark, so we create a helper function that applies all the settings we need:

```
void CustomArguments(benchmark::internal::Benchmark* b) {
 b->Arg(50)->Arg(10'000)->Arg(1'000'000)  // Input size
    ->MeasureProcessCPUTime()          // Measure all threads
    ->UseRealTime()                    // Clock on the wall
    ->Unit(benchmark::kMillisecond);    // Use ms
}
```

A few things to note about the custom arguments:

- We pass the values 50, 10,000, and 1,000,000 as arguments to the benchmark. They are used as the input size when creating the vectors in the `setup_fixture()` function. These values are accessed using `state.range(0)` in the test functions.
- By default, Google Benchmark only measures CPU time on the main thread. But since we are interested in the total amount of CPU time on all threads, we use `MeasureProcessCPUTime()`.
- Google Benchmark decides how many times each test needs to be repeated until a statistically stable result has been achieved. We want the library to use the clock-on-the-wall time for this rather than CPU time, and therefore we apply the setting `UseRealTime()`.

That's almost it. Finally, register the benchmarks and call main:

```
BENCHMARK(bm_sequential)->Apply(CustomArguments);
BENCHMARK(bm_parallel)->Apply(CustomArguments);
BENCHMARK_MAIN();
```

After compiling this code with optimizations turned on (using gcc with -O3), I executed this benchmark on a laptop using eight cores. The following table shows the results when using 50 elements:

Algorithm	CPU	Time	Speedup
`std::transform()`	0.02 ms	0.02 ms	0.25x
`par_transform_naive()`	0.17 ms	0.08 ms	

CPU is the total time spent on the CPU. *Time* is the wall-clock time, which is what we are most interested in. *Speedup* is the relative speedup when comparing the elapsed time of the sequential version with the parallel version (0.02/0.08 in this case).

Clearly, the sequential version outperforms the parallel algorithm for this small data set with only 50 elements. With 10,000 elements we really start to see the benefits of parallelization though:

Algorithm	CPU	Time	Speedup
`std::transform()`	0.89 ms	0.89 ms	4.5x
`par_transform_naive()`	1.95 ms	0.20 ms	

Finally, using 1,000,000 elements gives us even higher efficiency, as can be seen in the following table:

Algorithm	CPU	Time	Speedup
`std::transform()`	9071 ms	9092 ms	7.3x

`par_transform_naive()`	9782 ms	1245 ms	

The efficiency of the parallel algorithm in this last run is really high. It was executed on eight cores, so the efficiency is 7.3x/8 = 0.925. The results presented here (both the absolute execution time and the relative speedup) should not be relied upon too much. Among other things, the results depend on the computer architecture, the OS scheduler, and how much other work is currently running on the machine when performing the test. Nevertheless, the benchmarking results confirm a few important points discussed earlier:

- For small data sets, the sequential version `std::transform()` is much faster than the parallel version because of the overhead incurred by creating threads etc.
- The parallel version always uses more computational resources (CPU time) compared to `std::transform()`.
- For large data sets, the parallel version outperforms the sequential version when measuring wall-clock time. The speedup is over 7x when running on a machine with eight cores.

One reason for the high efficiency of our algorithm (at least on large data sets) is that the computational cost is evenly distributed, and each subtask is highly independent. This is not always the case, though.

Shortcomings of the naive implementation

The naive implementation might do a good job if each chunk of work has the same computational cost and the algorithm executes in an environment where no other application utilizes the hardware. However, this is rarely the case; rather, we want a good general-purpose parallel implementation that is both efficient and scalable.

The following illustrations show the problems we want to avoid. If the computational cost is not equivalent for each chunk, the implementation is limited to the chunk that takes the most time:

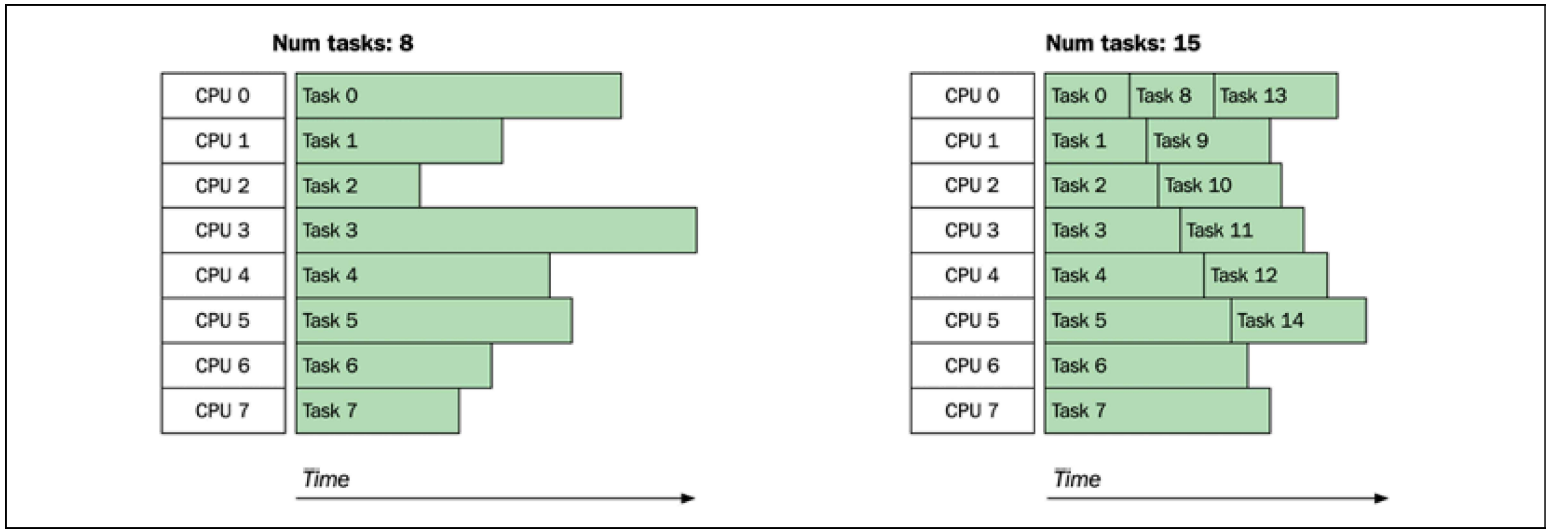

Figure 14.2: Possible scenarios where computation time is not proportional to chunk size

If the application and/or the operating system has other processes to handle, the operation will not process all chunks in parallel:

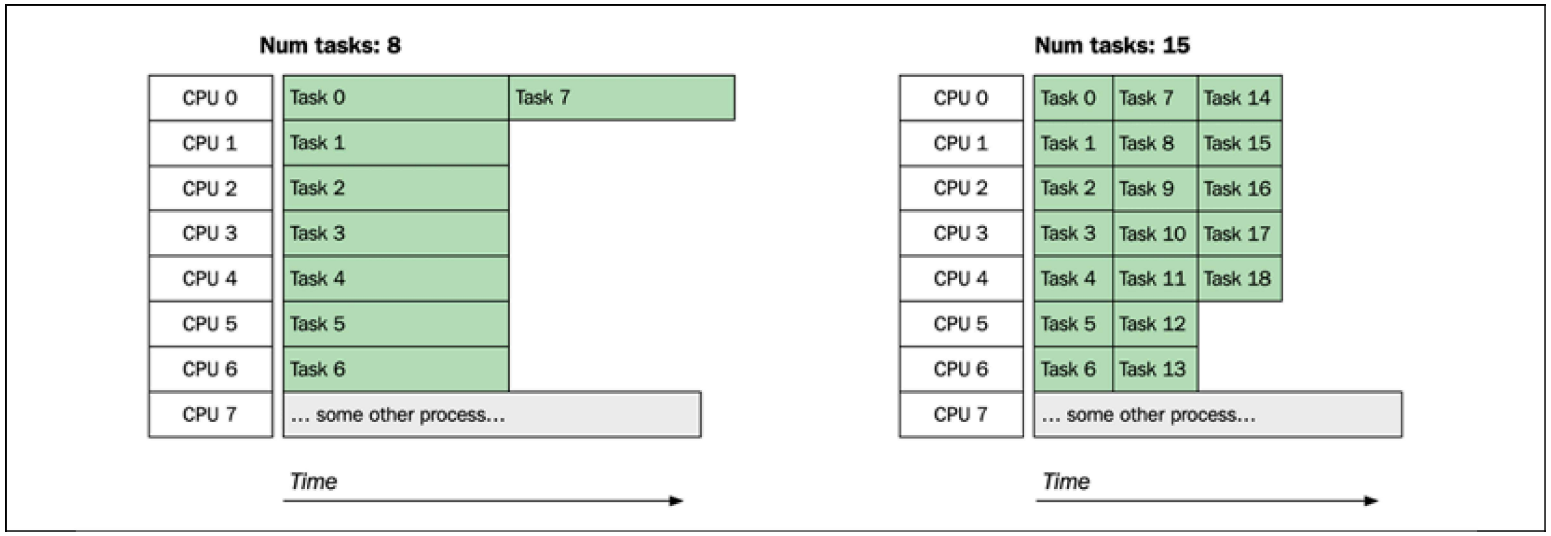

Figure 14.3: Possible scenarios where computation time is proportional to chunk size

As you can see in *Figure 14.3*, splitting the operation into smaller chunks makes the parallelization adjust to the current condition, avoiding single tasks that stall the whole operation.

Also note that the naive implementation was unsuccessful for small data sets. There are many ways to adjust the naive implementation to perform better. For instance, we could create more tasks and smaller tasks by multiplying the number of cores by some factor greater than 1. Or, to avoid significant overhead on small data sets, we could let the chunk size decide the number of tasks to create etc.

You now have the knowledge of how to implement and evaluate a simple parallel algorithm. We will not do any fine-tuning of the naive implementation; instead, I will show a different useful technique to use when implementing parallel algorithms.

Divide and conquer

An algorithm technique for dividing a problem into smaller subproblems is called **divide and conquer**. We will here implement another version of a parallel transform algorithm using divide and conquer. It works as follows: if the input range is smaller than a specified threshold, the range is processed; otherwise, the range is split into two parts:

- The first part is processed on a newly branched task
- The other part is recursively processed at the calling thread

The following illustration shows how the divide and conquer algorithm would recursively transform a range using the following data and parameters:

- Range size: 16
- Source range contains floats from 1.0 to 16.0
- Chunk size: 4
- Transformation function: `[](auto x) { return x*x; }`

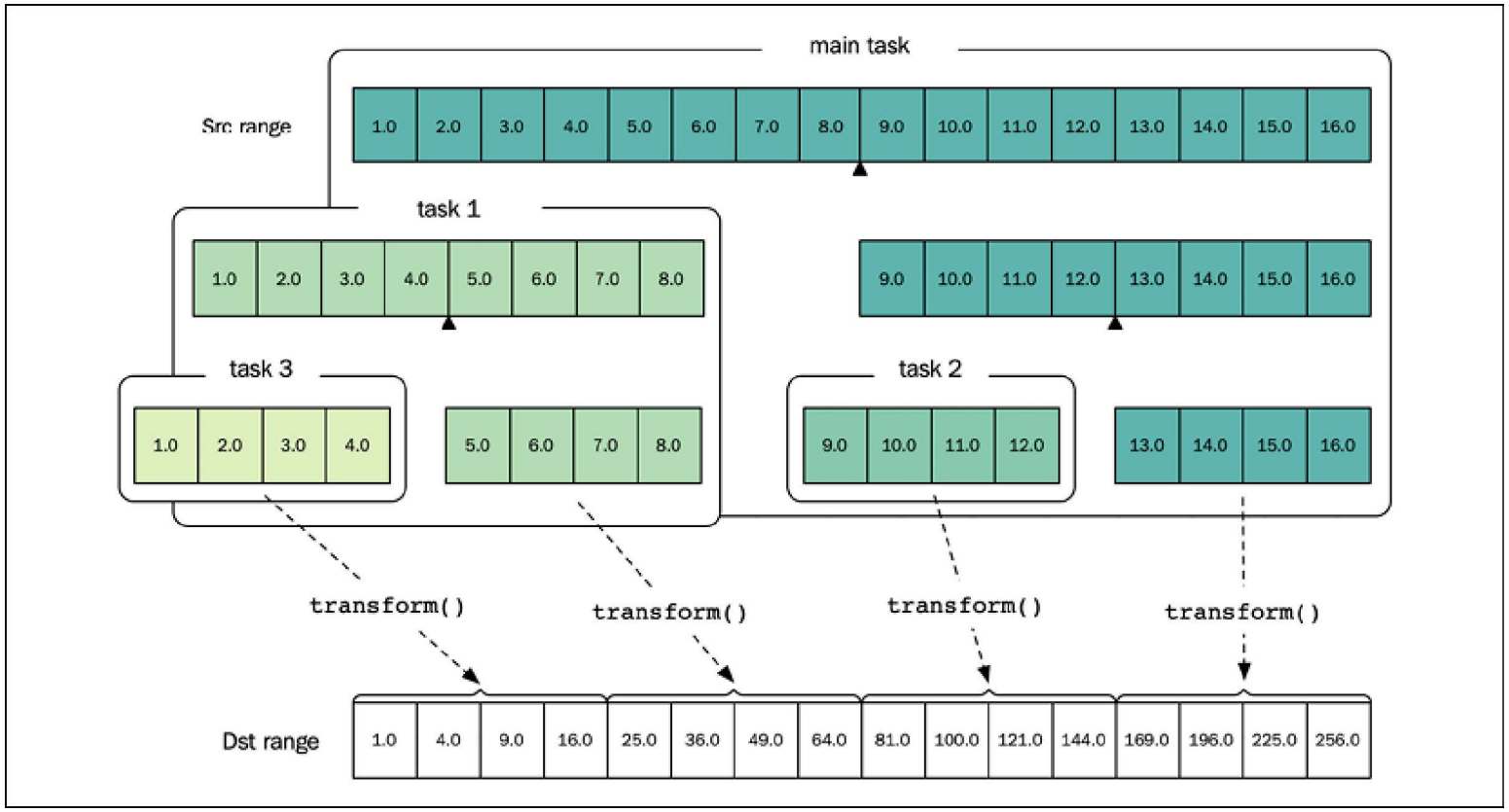

Figure 14.4: A range is divided recursively for parallel processing. The source array contains float values from 1.0 to 8.0. The destination array contains the transformed values.

In *Figure 14.4*, you can see that the main task spawns two asynchronous tasks (**task 1** and **task 2**) and finally transforms the last chunk in the range. **Task 1** spawns **task 3** and then transforms the remaining elements containing values 5.0, 6.0, 7.0, and 8.0. Let's head over to the implementation.

Implementation

Implementation-wise, it's quite a small bit of code. The incoming range is recursively split into two chunks; the first chunk is invoked as a new task, and the second chunk is recursively processed on the same task:

```
template <typename SrcIt, typename DstIt, typename Func>
auto par_transform(SrcIt first, SrcIt last, DstIt dst,
                   Func func, size_t chunk_sz) {
  const auto n = static_cast<size_t>(std::distance(first, last));
  if (n <= chunk_sz) {
    std::transform(first, last, dst, func);
    return;
  }
  const auto src_middle = std::next(first, n / 2);
  // Branch of first part to another task
  auto future = std::async(std::launch::async, [=, &func] {
    par_transform(first, src_middle, dst, func, chunk_sz);
  });
  // Recursively handle the second part
  const auto dst_middle = std::next(dst, n / 2);
  par_transform(src_middle, last, dst_middle, func, chunk_sz);
  future.wait();
}
```

Combining recursion with multithreading like this can take a while to get your head around. In the following examples, you will see that this pattern can be used when implementing more complicated algorithms as well. But first, let's see how it performs.

Performance evaluation

To evaluate our new version, we will modify the benchmark fixture by updating the transform function with a version that takes more time depending on the input value. The range of input values will be increased by filling the range using `std::iota()`. Doing this means the algorithms need to process jobs of different sizes. Here is the new `setup_fixture()` function:

```
auto setup_fixture(int n) {
  auto src = std::vector<float>(n);
  std::iota(src.begin(), src.end(), 1.0f);  // From 1.0 to n
  auto dst = std::vector<float>(src.size());
  auto transform_function = [](float v) {
    auto sum = v;
    auto n = v / 20'000;              // The larger v is,
    for (auto i = 0; i < n; ++i) {    // the more to compute
      sum += (i * i * i * sum);
    }
    return sum;
  };
  return std::tuple{src, dst, transform_function};
}
```

We can now try to find an optimal chunk size to be used by the divide-and-conquer algorithm by using an increasing parameter for the chunk size. It would also be interesting to see how our divide-and-conquer algorithm performs compared to the naive version on this new fixture, which needs to process jobs of different sizes. Here is the full code:

```
// Divide and conquer version
void bm_parallel(benchmark::State& state) {
```

```
  auto [src, dst, f] = setup_fixture(10'000'000);
  auto n = state.range(0);       // Chunk size is parameterized
  for (auto _ : state) {
    par_transform(src.begin(), src.end(), dst.begin(), f, n);
  }
}
// Naive version
void bm_parallel_naive(benchmark::State& state) {
  auto [src, dst, f] = setup_fixture(10'000'000);
  for (auto _ : state) {
    par_transform_naive(src.begin(), src.end(), dst.begin(), f);
  }
}
void CustomArguments(benchmark::internal::Benchmark* b) {
  b->MeasureProcessCPUTime()
    ->UseRealTime()
    ->Unit(benchmark::kMillisecond);
}
BENCHMARK(bm_parallel)->Apply(CustomArguments)
  ->RangeMultiplier(10)          // Chunk size goes from
  ->Range(1000, 10'000'000);     // 1k to 10M
BENCHMARK(bm_parallel_naive)->Apply(CustomArguments);
BENCHMARK_MAIN();
```

The following diagram reveals the results I achieved when running the tests on macOS using an Intel Core i7 CPU with eight cores:

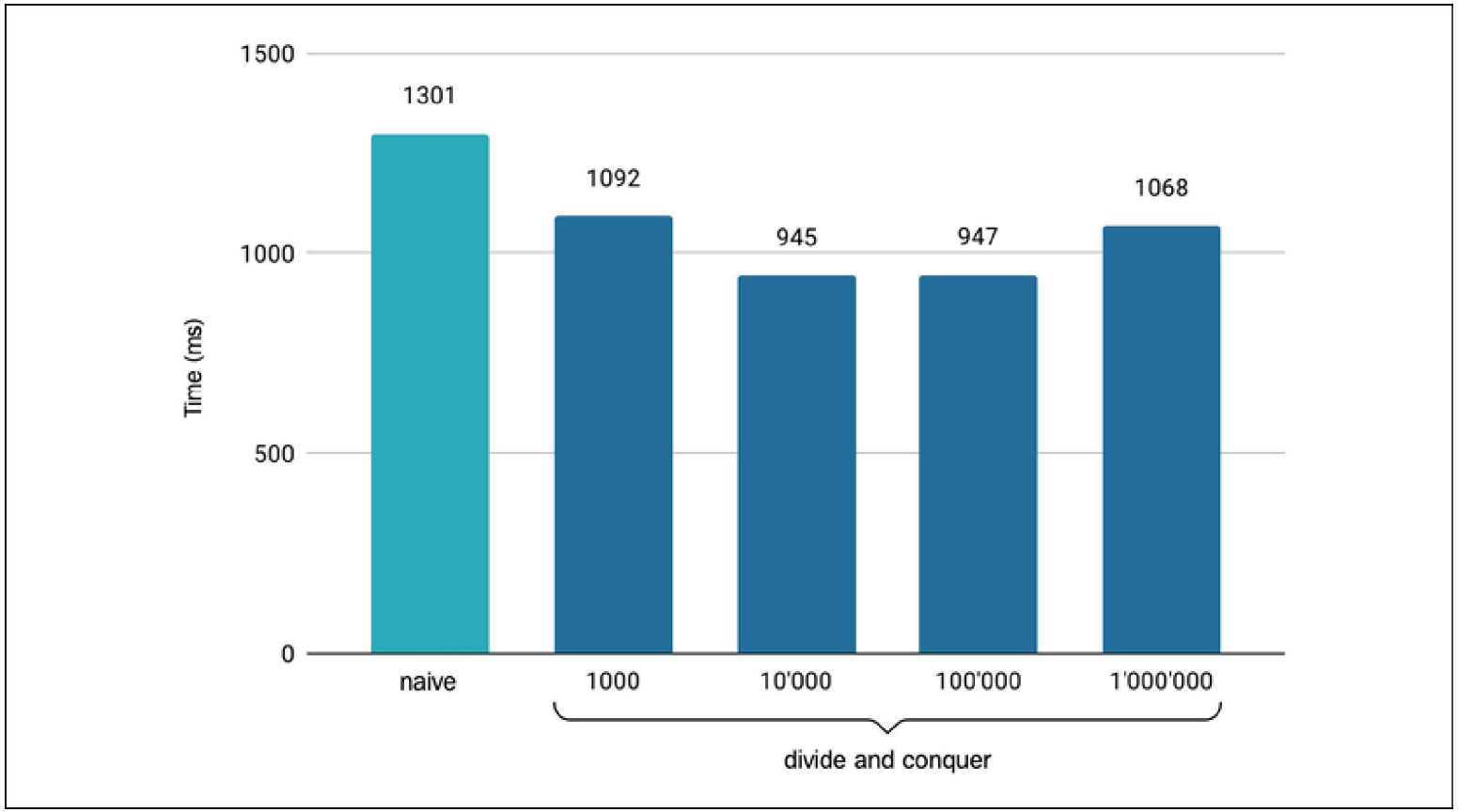

Figure 14.5: Comparison between our naive algorithm and the divide-and-conquer algorithm using different chunk sizes

The best efficiency was achieved when using chunk sizes of around 10,000 elements, which creates 1,000 tasks. With larger chunks, the performance is bottlenecked in the time it takes to process the final chunks, whereas too small chunks result in too much overhead in creating and invoking tasks compared to the computation.

A takeaway from this example is that the performance penalty of scheduling 1,000 smaller tasks rather than a few big ones isn't a problem here. It would be possible to restrict the number of threads using a thread pool, but `std::async()` seems to work fairly well in this scenario. A generic implementation would opt for using a fairly large number of tasks rather than trying to match the exact number of cores.

Finding optimal values for chunk size and the number of tasks is a real problem when implementing parallel algorithms. As you can see, it depends on many variables and also whether you optimize for latency or throughput. The best way to gain insights is to measure in the environment that your algorithms are supposed to run.

Now that you have learned how to implement a parallel transform algorithm using divide and conquer, let's see how the same technique can be applied to other problems.

Implementing parallel std::count_if()

A nice thing with divide and conquer is that it can be applied to many problems. We can easily use the same technique to implement a parallel version of `std::count_if()`, with the difference being that we need to accumulate the returned value, like this:

```
template <typename It, typename Pred>
auto par_count_if(It first, It last, Pred pred, size_t chunk_sz) {
  auto n = static_cast<size_t>(std::distance(first, last));
  if (n <= chunk_sz)
    return std::count_if(first, last, pred);
  auto middle = std::next(first, n/2);
  auto fut = std::async(std::launch::async, [=, &pred] {
    return par_count_if(first, middle, pred, chunk_sz);
```

```
  });
  auto num = par_count_if(middle, last, pred, chunk_sz);
  return num + fut.get();
}
```

As you can see, the only difference here is that we need to sum the result at the end of the function. If you want to have the chunk size depend on the number of cores, you can easily wrap the `par_count_if()` in an outer function:

```
template <typename It, typename Pred>
auto par_count_if(It first, It last, Pred pred) {
  auto n = static_cast<size_t>(std::distance(first, last));
  auto n_cores = size_t{std::thread::hardware_concurrency()};
  auto chunk_sz = std::max(n / n_cores * 32, size_t{1000});

  return par_count_if(first, last, pred, chunk_sz);
}
```

The magic number 32 here is a somewhat arbitrary factor that will give us more chunks and smaller chunks if we are given a large input range. As usual, we would need to measure the performance to come up with a good constant here. Let's now move on and try to tackle a more complicated parallel algorithm.

Implementing parallel std::copy_if()

We've had a look at `std::transform()` and `std::count_if()`, which are quite easy to implement both sequentially and in parallel. If we take another algorithm that is easily implemented sequentially,

`std::copy_if()`, things get a lot harder to perform in parallel.

Sequentially, implementing `std::copy_if()` is as easy as this:

```
template <typename SrcIt, typename DstIt, typename Pred>
auto copy_if(SrcIt first, SrcIt last, DstIt dst, Pred pred) {
 for (auto it = first; it != last; ++it) {
  if (pred(*it)) {
   *dst = *it;
   ++dst;
  }
 }
 return dst;
}
```

To demonstrate how it can be used, consider the following example where we have a range that contains a sequence of integers and we want to copy only the odd integers into another range:

```
const auto src = {1, 2, 3, 4};
auto dst = std::vector<int>(src.size(), -1);
auto new_end = std::copy_if(src.begin(), src.end(), dst.begin(),
                            [](int v) { return (v % 2) == 1; });
// dst is {1, 3, -1, -1}
dst.erase(new_end, dst.end()); // dst is now {1, 3}
```

Now, if we want to make a parallel version of `copy_if()`, we immediately run into problems as we cannot write to the destination iterator concurrently. Here is a failed attempt with undefined behavior, since

both tasks will write to the same position in the destination range:

```
// Warning: Undefined behavior
template <typename SrcIt, typename DstIt, typename Func>
auto par_copy_if(SrcIt first, SrcIt last, DstIt dst, Func func) {
 auto n = std::distance(first, last);
 auto middle = std::next(first, n / 2);
 auto fut0 = std::async([=]() {
  return std::copy_if(first, middle, dst, func); });
 auto fut1 = std::async([=]() {
  return std::copy_if(middle, last, dst, func); });
 auto dst0 = fut0.get();
 auto dst1 = fut1.get();
 return *std::max(dst0, dst1); // Just to return something...
}
```

We now have two simple approaches: either we synchronize the index we write to (by using an atomic/lock-free variable), or we split the algorithm into two parts. We will explore both approaches next.

Approach 1: Use a synchronized write position

The first approach we might consider is to synchronize the write position by using an atomic `size_t` and the `fetch_add()` member function, as you learned about in *Chapter 11*, *Concurrency*. Whenever a thread tries to write a new element, it fetches the current index and adds one atomically; thus, each value is written to a unique index.

In our code, we will split the algorithm into two functions: an inner function and an outer function. The atomic write index will be defined in the outer function, whereas the main part of the algorithm will be implemented in the inner function.

Inner function

The inner function requires an atomic `size_t` that synchronizes the write positions. As the algorithm is recursive, it cannot store the atomic `size_t` itself; it requires an outer function to invoke the algorithm:

```
template <typename SrcIt, typename DstIt, typename Pred>
void inner_par_copy_if_sync(SrcIt first, SrcIt last, DstIt dst,
                            std::atomic_size_t& dst_idx,
                            Pred pred, size_t chunk_sz) {
 const auto n = static_cast<size_t>(std::distance(first, last));
 if (n <= chunk_sz) {
  std::for_each(first, last, [&](const auto& v) {
   if (pred(v)) {
    auto write_idx = dst_idx.fetch_add(1);
    *std::next(dst, write_idx) = v;
   }
  });
  return;
 }
 auto middle = std::next(first, n / 2);
 auto future = std::async([first, middle, dst, chunk_sz, &pred, &dst_idx] {
  inner_par_copy_if_sync(first, middle, dst, dst_idx, pred, chunk_sz);
 });
 inner_par_copy_if_sync(middle, last, dst, dst_idx, pred, chunk_sz);
```

```
  future.wait();
}
```

This is still a divide-and-conquer algorithm and hopefully you will start to see the pattern we are using. The atomic update of the write index `dst_idx` ensures that multiple threads never write to the same index in the destination sequence.

Outer function

The outer function, called from the client code, is simply a placeholder for the atomic `size_t`, which is initialized to zero. The function then initializes the inner function, which parallelizes the code further:

```
template <typename SrcIt, typename DstIt, typename Pred>
auto par_copy_if_sync(SrcIt first,SrcIt last,DstIt dst,
                      Pred p, size_t chunk_sz) {
  auto dst_write_idx = std::atomic_size_t{0};
  inner_par_copy_if_sync(first, last, dst, dst_write_idx, p, chunk_sz);
  return std::next(dst, dst_write_idx);
}
```

Once the inner function returns, we can use `dst_write_idx` to compute the end iterator of the destination range. Let's now have a look at the other approach to solve the same problem.

Approach 2: Split the algorithm into two parts

The second approach is to split the algorithm into two parts. First, the conditional copying is performed in parallel chunks, and then the resulting sparse range is squeezed to a continuous range.

Part one – Copy elements in parallel into the destination range

The first part copies the elements in chunks, resulting in the sparse destination array illustrated in *Figure 14.6*. Each chunk is conditionally copied in parallel, and the resulting range iterators are stored in `std::future` objects for later retrieval:

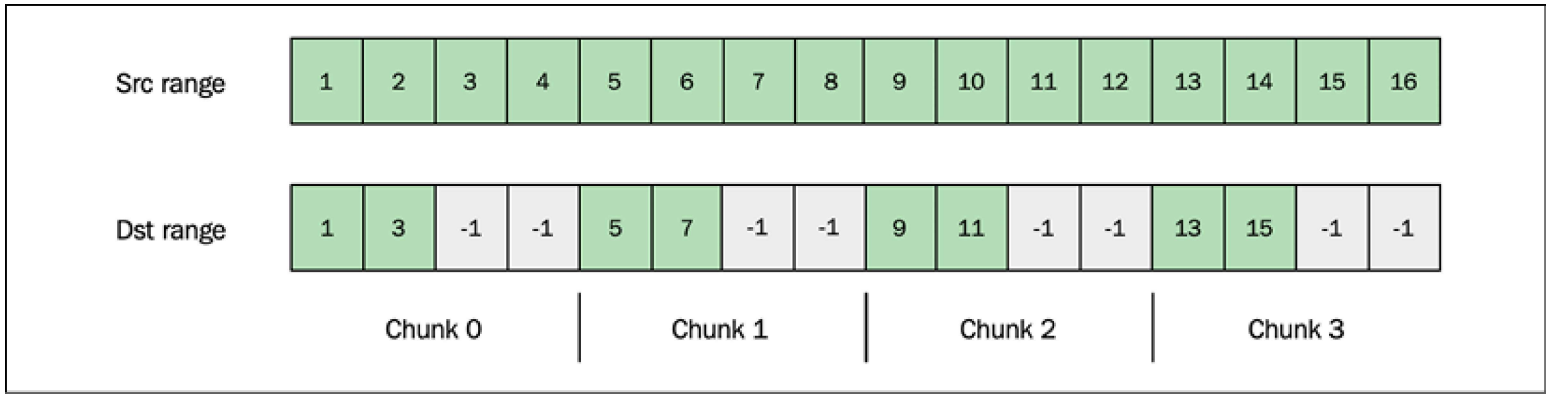

Figure 14.6: The sparse destination range after the first step of the conditional copying

The following code implements the first half of the algorithm:

```
template <typename SrcIt, typename DstIt, typename Pred>
auto par_copy_if_split(SrcIt first, SrcIt last, DstIt dst,
                       Pred pred, size_t chunk_sz) -> DstIt {
  auto n = static_cast<size_t>(std::distance(first, last));
  auto futures = std::vector<std::future<std::pair<DstIt, DstIt>>>{};
  futures.reserve(n / chunk_sz);
  for (auto i = size_t{0}; i < n; i += chunk_sz) {
    const auto stop_idx = std::min(i + chunk_sz, n);
    auto future = std::async([=, &pred] {
```

```
    auto dst_first = dst + i;
    auto dst_last = std::copy_if(first+i, first+stop_idx,                    dst_first, pred);
    return std::make_pair(dst_first, dst_last);
   });
   futures.emplace_back(std::move(future));
  }
  // To be continued ...
```

We have now copied the elements (that should be copied) into the sparse destination range. It's time to fill the gaps by moving the elements to the left in the range.

Part two - Move the sparse range sequentially into a continuous range

When the sparse range is created, it is merged using the resulting value from each `std::future`. The merge is performed sequentially as the parts overlap:

```
  // ...continued from above...
  // Part #2: Perform merge of resulting sparse range sequentially
  auto new_end = futures.front().get().second;
  for (auto it = std::next(futures.begin()); it != futures.end(); ++it)  {
   auto chunk_rng = it->get();
   new_end = std::move(chunk_rng.first, chunk_rng.second, new_end);
  }
  return new_end;
 } // end of par_copy_if_split
```

This second part of the algorithm that moves all the subranges to the beginning of the range is illustrated in the following image:

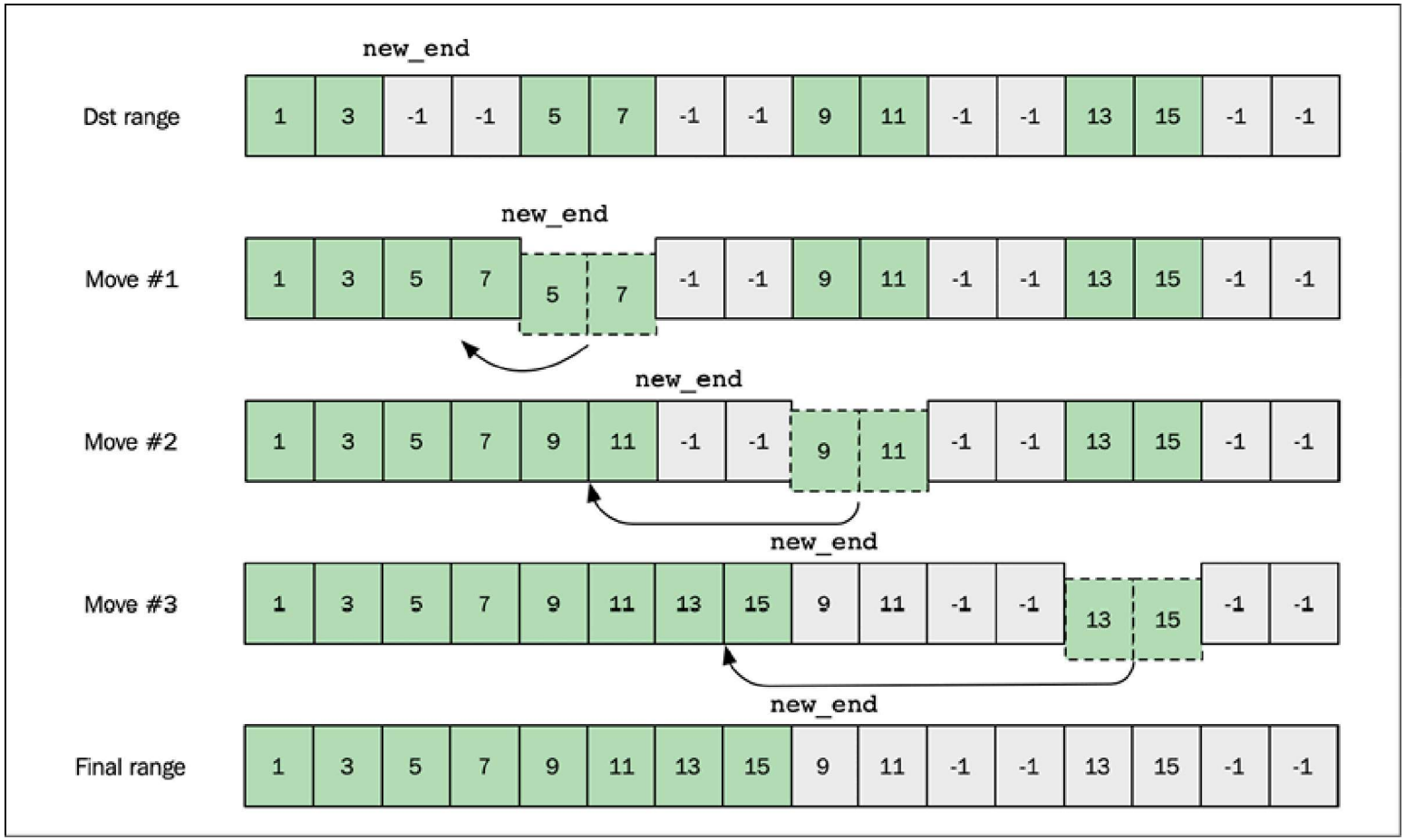

Figure 14.7: Merging a sparse range into a continuous range

With two algorithms solving the same problem, it's time to see how they measure up.

Performance evaluation

The performance boost from using this parallelized version of `copy_if()` is heavily dependent on how expensive the predicate is. Therefore, we use two different predicates in our benchmark with different computational costs. Here is the *inexpensive* predicate:

```
auto is_odd = [](unsigned v) {
  return (v % 2) == 1;
};
```

The more *expensive* predicate checks whether its argument is a prime number:

```
auto is_prime = [](unsigned v) {
  if (v < 2) return false;
  if (v == 2) return true;
  if (v % 2 == 0) return false;
  for (auto i = 3u; (i * i) <= v; i+=2) {
    if ((v % i) == 0) {
      return false;
    }
  }
  return true;
};
```

Note, this is not a particularly optimal way to implement `is_prime()`, and is only used here for the purposes of the benchmark.

The benchmarking code is not spelled out here but is included in the accompanying source code. Three algorithms are compared: `std::copy_if()`, `par_copy_if_split()`, and `par_copy_if_sync()`. The following graph shows the results as measured using an Intel Core i7 CPU. The parallel algorithms use a chunk size of 100,000 in this benchmark.

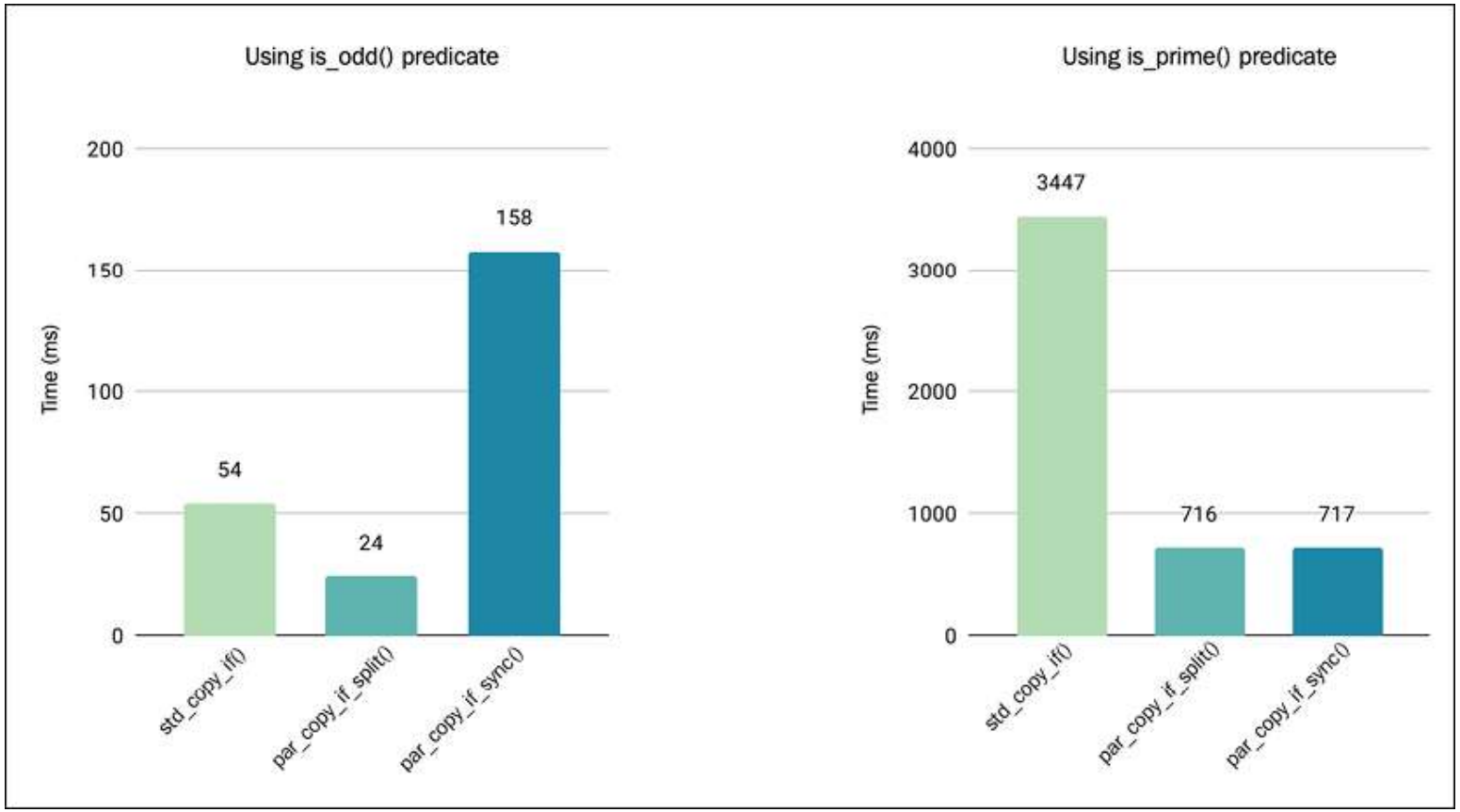

Figure 14.8: Conditional copy strategies versus computation time

The most obvious observation when measuring the performance is how ridiculously slow the synchronized version `par_copy_if_sync()` is when using the inexpensive `is_odd()` predicate. The disastrous per-

formance is actually not due to the atomic write index; rather, it is because the cache mechanism of the hardware is trashed due to several threads writing to the same cache line (as you learned about in *Chapter 7*, *Memory Management*).

So, with this knowledge, we understand now why `par_copy_if_split()` performs better. On the inexpensive predicate, `is_odd()`, `par_copy_if_split()` is about 2x faster than `std::copy_if()`, but with the expensive `is_prime()`, the efficiency increases to almost 5x. The increased efficiency is a result of spending most of the computations in the first part of the algorithm, which executes in parallel.

You should now have a grasp of some techniques that can be used for parallelizing an algorithm. These new insights will help you understand the requirements and expectations when using parallel algorithms from the standard library.

Parallel standard library algorithms

As of C++17, the standard library has been extended with parallel versions of most, but not all, algorithms. Changing your algorithms to allow for parallel execution is simply a matter of adding a parameter that tells the algorithm which parallel execution policy to use.

As stressed earlier in this book, if your code base is based upon standard library algorithms, or at least if you have the habit of writing C++ by using algorithms, you will get an instant performance boost almost for free by adding an execution policy where suitable:

```
auto v = std::vector<std::string>{
  "woody", "steely", "loopy", "upside_down"
};
```

```
// Parallel sort
std::sort(std::execution::par, v.begin(), v.end());
```

Once you specify an execution policy, you are in the realm of parallel algorithms, which have some notable differences compared to their original sequential versions. Firstly, the minimum iterator category requirements change from input iterators to forward iterators. Secondly, exceptions thrown by your code (from copy constructors or function objects passed to the algorithm) never reach you. Instead, the algorithm is required to call `std::terminate()`. Thirdly, the complexity guarantees (both time and memory) of the algorithm might be relaxed because of the added complexity of parallel implementations.

When using a parallel version of the standard library algorithms, you specify an execution policy that states how an algorithm is allowed to parallelize the execution. An implementation may decide to execute the algorithm sequentially, though. If you compare the efficiency and scalability of parallel algorithms in different standard library implementations, you can expect to see big differences.

Execution policies

An **execution policy** informs an algorithm if and how the execution can be parallelized. There are four default execution policies included in the parallel extensions of the standard library. Compilers and third-party libraries can extend these policies for certain hardware and conditions. For example, it's already possible to use the parallel power of the modern graphics card from standard library algorithms using vendor-specific policies.

The execution policies are defined in the header `<execution>` and reside in the namespace `std::execution`. There are currently four distinct tag types, one for each execution policy. The types cannot be instantiated by you; instead, there is one predefined object per type. For instance, the parallel ex-

ecution policy has a type called `std::execution::parallel_policy` and the predefined instance of this type is named `std::execution::par`. The reason there is one *type* per policy (rather than one type with multiple predefined instances) is so that the policies you provide can be distinguished at compile time by the library.

Sequenced policy

The sequenced execution policy, `std::execution::seq`, makes the algorithm execute sequentially with no parallelism, similar to how the algorithms without the extra execution policy argument would run. However, whenever you specify an execution policy, it means that you are using a version of the algorithm with relaxed complexity guarantees and stricter iterator requirements; it also assumes that the code you provide doesn't throw exceptions, or the algorithm will call `std::terminate()`.

Parallel policy

The parallel execution policy, `std::execution::par`, can be considered the standard execution policy for parallel algorithms. The code you provide to the algorithm needs to be thread safe. A way to understand this requirement is to think about the loop body in the sequential version of the algorithm you are about to use. For example, think about the sequential version of `copy_if()`, which we spelled out like this earlier in the chapter:

```
template <typename SrcIt, typename DstIt, typename Pred>
auto copy_if(SrcIt first, SrcIt last, DstIt dst, Pred pred) {
  for (auto it = first; it != last; ++it)
  {                       // Start of loop body
    if (pred(*it)) {        // Call predicate
      *dst = *it;           // Copy construct
```

```
      ++dst;
    }
  }                    // End of loop body
  return dst;
}
```

In this algorithm, the code inside the loop body will call the predicate you provided and invoke the copy assignment operator on the elements in the range. If you pass `std::execution::par` to `copy_if()`, it is your responsibility to guarantee that these parts are thread safe and can safely be executed in parallel.

Let's look at an example where we provide unsafe code and then see what we can do about it. Assume we have a vector of strings:

```
auto v = std::vector<std::string>{"Ada", "APL" /* ... */};
```

If we want to compute the total size of all strings in the vector using a parallel algorithm, an inadequate way to do this would be to use `std::for_each()`, like this:

```
auto tot_size = size_t{0};
std::for_each(std::execution::par, v.begin(), v.end(),
              [&](const auto& s) {
  tot_size += s.size(); // Undefined behavior, data race!
});
```

Since the body of the function object is not thread safe (as it updates a shared variable from multiple threads), this code exhibits undefined behavior. We could, of course, protect the `tot_size` variable with a `std::mutex`, but that would defeat the whole purpose of executing this code in parallel, since the mutex would only allow one thread at a time to enter the body. Using an `std::atomic` data type would be another option, but that could also degrade the efficiency.

The solution here is to *not* use `std::for_each()` for this problem at all. Instead, we can use `std::transform_reduce()` or `std::reduce()`, which are tailor-made for this kind of job. Here is how you would do it using `std::reduce()`:

```
auto tot_size = std::reduce(std::execution::par, v.begin(), v.end(),                    size_t{0}, [](auto i, const auto& s) {
  return i + s.size();   // OK! Thread safe
});
```

By getting rid of the mutable reference inside the lambda, the body of the lambda is now thread safe. The `const` reference to the `std::string` objects is fine, because it never mutates any string objects and therefore doesn't introduce any data races.

Normally, the code you pass to an algorithm is thread safe unless your function objects capture objects by reference or have other side effects such as writing to files.

Unsequenced policy

The unsequenced policy was added in C++20. It tells the algorithm that the loop is allowed to be vectorized using, for example, SIMD instructions. In practice, this means that you cannot use any synchronization primitives in the code you pass to the algorithm, since this could result in deadlocks.

To understand how a deadlock can occur, we will get back to the previous inadequate example when counting the total size of all strings in a vector. Assume that, instead of using `std::reduce()`, we protect the `tot_size` variable by adding a mutex, like this:

```
auto v = std::vector<std::string>{"Ada", "APL" /* ... */ };
auto tot_size = size_t{0};
auto mut = std::mutex{};
std::for_each(std::execution::par, v.begin(), v.end(),
              [&](const auto& s) {
   auto lock = std::scoped_lock{mut}; // Lock
   tot_size += s.size();
 }                                    // Unlock
);
```

This code is now safe to execute using `std::execution::par`, but it is very inefficient. If we were to change the execution policy to `std::execution::unseq`, the result would not only be an inefficient program but also a program that runs the risk of deadlocking!

The unsequenced execution policy tells the algorithm that it may reorder the instruction of our code in a way that is normally not allowed by the optimizing compiler.

For the algorithm to benefit from vectorization, it needs to read multiple values from the input range, and then apply SIMD instructions to multiple values at once. Let's analyze what two iterations in the loop of `for_each()` could look like, with and without reorderings. Here are two loop iterations without any reorderings:

```
{ // Iteration 1
  const auto& s = *it++;
  mut.lock();
  tot_size += s.size();
  mut.unlock();
}
{ // Iteration 2
  const auto& s = *it++;
  mut.lock();
  tot_size += s.size();
  mut.unlock();
}
```

The algorithm is allowed to merge these two iterations in the following way:

```
{ // Iteration 1 & 2 merged
  const auto& s1 = *it++;
  const auto& s2 = *it++;
  mut.lock();
  mut.lock();              // Deadlock!
  tot_size += s1.size();   // Replace these operations
  tot_size += s2.size();   // with vectorized instructions
  mut.unlock();
  mut.unlock();
}
```

Trying to execute this code on the same thread will deadlock because we are trying to lock the very same mutex twice consecutively. In other words, when using the `std::execution::unseq` policy, you must make sure that the code you provide to the algorithm doesn't acquire any locks.

Note that the optimizing compiler is free to vectorize your code anytime. However, in those cases, it's up to the compiler to guarantee that the vectorization doesn't change the meaning of the program, just like any other optimizations that the compiler and hardware are allowed to perform. The difference here, when explicitly providing the `std::execute::unseq` policy to an algorithm, is that *you* guarantee that the code you provide is safe to vectorize.

Parallel unsequenced policy

The parallel unsequenced policy, `std::execution::par_unseq`, executes the algorithm in parallel like the parallel policy, with the addition that it may also vectorize the loop.

Apart from the four standard execution policies, standard library vendors can provide you with additional policies with custom behavior and put other constraints on the input. For example, the Intel Parallel STL library defines four custom execution policies that only accept random access iterators.

Exception handling

If you provide one of the four standard execution policies to an algorithm, your code must not throw exceptions, or the algorithm will call `std::terminate()`. This is a big difference from the normal single-threaded algorithms, which always propagate exceptions back to the caller:

```
auto v = {1, 2, 3, 4};
auto f = [](auto) { throw std::exception{}; };
```

```
try {
  std::for_each(v.begin(), v.end(), f);
} catch (...) {
  std::cout << "Exception caught\n";
}
```

Running the same code with an execution policy results in a call to `std::terminate()`:

```
try {
  std::for_each(std::execution::seq, v.begin(), v.end(), f);
} catch (...) {
  // The thrown std:::exception never reaches us.
  // Instead, std::terminate() has been called
}
```

You might think that this means the parallel algorithms are declared `noexcept`, but that's not the case. Many parallel algorithms need to allocate memory, and therefore the standard parallel algorithms themselves are allowed to throw `std::bad_alloc`.

It should also be said that execution policies provided by other libraries may handle exceptions in a different way.

Now, we will move on to discuss some of the algorithms that were added and modified when the parallel algorithms were first introduced in C++17.

Additions and changes to parallel algorithms

Most algorithms in the standard library are available as parallel versions straight out the box. However, there are some noteworthy exceptions, including `std::accumulate()` and `std::for_each()`, as their original specifications required in-order execution.

std::accumulate() and std::reduce()

The `std::accumulate()` algorithm cannot be parallelized as it must be executed in the order of the elements, which is not possible to parallelize. Instead, a new algorithm called `std::reduce()` has been added, which works just like `std::accumulate()` with the exception that it is executed unordered.

With commutative operations, their results are the same, as the order of accumulation doesn't matter. In other words, given a range of integers:

```
const auto r = {1, 2, 3, 4};
```

accumulating them by addition or multiplication:

```
auto sum =
  std::accumulate(r.begin(), r.end(), 0, std::plus<int>{});

auto product =
  std::accumulate(r.begin(), r.end(), 1, std::multiplies<int>{});
```

would yield the same result as invoking `std::reduce()` instead of `std::accumulate()`, as both the addition and multiplication of integers are commutative. For example:

$$(1+2+3+4) = (3+1+4+2) \; and \; (1 \cdot 2 \cdot 3 \cdot 4) = (3 \cdot 2 \cdot 1 \cdot 4)$$

But, if the operation is not commutative, the result is *non-deterministic* since it depends on the order of the arguments. For example, if we were to accumulate a list of strings as follows:

```
auto v = std::vector<std::string>{"A", "B", "C"};
auto acc = std::accumulate(v.begin(), v.end(), std::string{});
std::cout << acc << '\n'; // Prints "ABC"
```

this code will always produce the string `"ABC"`. But, by using `std::reduce()`, the characters in the resulting string could be in any order because string concatenation is not commutative. In other words, the string `"A" + "B"` is not equal to `"B" + "A"`. Therefore, the following code using `std::reduce()` might produce different results:

```
auto red = std::reduce(v.begin(), v.end(), std::string{});
std::cout << red << '\n';
// Possible output: "CBA" or "ACB" etc
```

An interesting point related to performance is that floating-point math is not commutative. By using `std::reduce()` on floating-point values, the results may vary, but it also means that `std::reduce()` is potentially much faster than `std::accumulate()`. This is because `std::reduce()` is allowed to reorder operations and utilize SIMD instructions in a way that `std::accumulate()` isn't allowed to do when using strict floating-point math.

std::transform_reduce()

As an addition to the standard library algorithms, `std::transform_reduce()` has also been added to the `<numeric>` header. It does exactly what it says: it transforms a range of elements as `std::transform()` and then applies a function object. This accumulates them out of order, like `std::reduce()`:

```
auto v = std::vector<std::string>{"Ada","Bash","C++"};
auto num_chars = std::transform_reduce(
  v.begin(), v.end(), size_t{0},
  [](size_t a, size_t b) { return a + b; },     // Reduce
  [](const std::string& s) { return s.size(); } // Transform
);
// num_chars is 10
```

Both `std::reduce()` and `std::transform_reduce()` were added to C++17 when parallel algorithms were introduced. Another necessary change was to adjust the return type of `std::for_each()`.

std::for_each()

A somewhat rarely used property of `std::for_each()` is that it returns the function object passed into it. This makes it possible to use `std::for_each()` to accumulate values inside a stateful function object. The following examples demonstrate a possible use case:

```
struct Func {
  void operator()(const std::string& s) {
    res_ += s;
  };
  std::string res_{};    // State
};
```

```
auto v = std::vector<std::string>{"A", "B", "C"};
auto s = std::for_each(v.begin(), v.end(), Func{}).res_;
// s is "ABC"
```

This usage is similar to what we can achieve using `std::accumulate()` and therefore also exhibits the same problem when trying to parallelize it: executing the function object out of order would yield non-deterministic results as the invocation order is undefined. Consequently, the parallel version of `std::for_each()` simply returns `void`.

Parallelizing an index-based for-loop

Even though I recommend using algorithms, sometimes a raw, index-based `for`-loop is required for a specific task. The standard library algorithms provide an equivalent of a range-based `for`-loop by including the algorithm `std::for_each()` in the library.

However, there is no algorithm equivalent of an index-based `for`-loop. In other words, we cannot easily parallelize code like this by simply adding a parallel policy to it:

```
auto v = std::vector<std::string>{"A", "B", "C"};
for (auto i = 0u; i < v.size(); ++i) {
  v[i] += std::to_string(i+1);
}
// v is now { "A1", "B2", "C3" }
```

But let's see how we can build one by combining algorithms. As you will have already concluded, implementing parallel algorithms is complicated. But in this case, we will build a `parallel_for()` algorithm using

`std::for_each()` as a building block, thus leaving the complex parallelism to `std::for_each()`.

Combining std::for_each() with std::views::iota()

An index-based `for`-loop based on a standard library algorithm can be created by combining `std::for_each()` with `std::views::iota()` from the ranges library (see *Chapter 6*, *Ranges and Views*). This is how it would look:

```
auto v = std::vector<std::string>{"A", "B", "C"};
auto r = std::views::iota(size_t{0}, v.size());
std::for_each(r.begin(), r.end(), [&v](size_t i) {
  v[i] += std::to_string(i + 1);
});
// v is now { "A1", "B2", "C3" }
```

This can then be further parallelized by using the parallel execution policy:

```
std::for_each(std::execution::par, r.begin(), r.end(), [&v](size_t i) {
  v[i] += std::to_string(i + 1);
});
```

As stated earlier, we have to be very careful when passing references to a lambda that will be invoked from multiple threads like this. By only accessing vector elements via the unique index `i`, we avoid introducing data races when mutating the strings in the vector.

Simplifying construction via a wrapper

In order to iterate the indices with a neat syntax, the previous code is wrapped into a utility function named `parallel_for()`, as shown here:

```
template <typename Policy, typename Index, typename F>
auto parallel_for(Policy&& p, Index first, Index last, F f) {
  auto r = std::views::iota(first, last);
  std::for_each(p, r.begin(), r.end(), std::move(f));
}
```

The `parallel_for()` function template can then be used directly like this:

```
auto v = std::vector<std::string>{"A", "B", "C"};
parallel_for(std::execution::par, size_t{0}, v.size(),
             [&](size_t i) { v[i] += std::to_string(i + 1); });
```

As the `parallel_for()` is built upon `std::for_each()`, it accepts any policy that `std::for_each()` accepts.

We will wrap this chapter up with a short introductory overview of GPUs and how they can be used for parallel programming, now and in the future.

Executing algorithms on the GPU

Graphics processing units (**GPUs**) were originally designed and used for processing points and pixels for computer graphics rendering. Briefly, what the GPUs did was retrieve buffers of pixel data or vertex data,

perform a simple operation on each buffer individually, and store the result in a new buffer (to eventually be displayed).

Here are some examples of simple, independent operations that could be executed on the GPU at an early stage:

- Transform a point from world coordinates to screen coordinates
- Perform a lighting calculation at a specific point (by lighting calculation, I am referring to calculating the color of a specific pixel in an image)

As these operations could be performed in parallel, the GPUs were designed for executing small operations in parallel. Later on, these graphics operations became programmable, although the programs were written in terms of computer graphics (that is, the memory reads were done in terms of reading colors from a texture, and the result was always written as a color to a texture). These programs are called **shaders**.

Over time, more shader-type programs were introduced, and shaders gained more and more low-level options, such as reading and writing raw values from buffers instead of color values from textures.

Technically, a CPU commonly consists of a few general-purpose cached cores, whereas a GPU consists of a huge number of highly specialized cores. This means that parallel algorithms that scale well are highly suitable to execute on a GPU.

GPUs have their own memory and before an algorithm can execute on the GPU, the CPU needs to allocate memory in the GPU memory and copy data from main memory to GPU memory. The next thing that happens is that the CPU launches a routine (also called a kernel) on the GPU. Finally, the CPU copies data back from the GPU memory into the main memory, making it accessible for "normal" code executing on

the CPU. The overhead incurred by copying data back and forth between the CPU and the GPU is one of the reasons why GPUs are more suitable for batch processing tasks where throughput is more important than latency.

There are several libraries and abstraction layers available today that make GPU programming accessible from C++, but standard C++ offers nearly nothing in this area. However, the parallel execution policies `std::execution::par` and `std::execution::par_unseq` allow compilers to move the execution of standard algorithms from the CPU to the GPU. One example of this is NVC++, the NVIDIA HPC compiler. It can be configured to compile standard C++ algorithms for execution on NVIDIA GPUs.

If you want to learn more about the current status of C++ and GPU programming, I highly recommend the talk *GPU Programming with modern C++* by Michael Wong (**https://accu.org/video/spring-2019-day-3/wong/**) from the ACCU 2019 conference.

Summary

In this chapter, you have learned about the complexity of handcrafting an algorithm to execute in parallel. You also now know how to analyze, measure, and tune the efficiency of parallel algorithms. The insights you gained while learning about parallel algorithms will have deepened your understanding of the requirements and the behaviors of the parallel algorithms found in the C++ standard library. C++ comes with four standard execution policies, which can be extended by compiler vendors. This opens up the door for utilizing the GPU for standard algorithms. The next C++ standard, C++23, will most likely add increased support for parallel programming on the GPU.

You have now reached the end of the book. Congratulations! Performance is an important aspect of code quality. But too often, performance comes at the expense of other quality aspects, such as readability,

maintainability, and correctness. Mastering the art of writing efficient and clean code requires practical training. My hope is that you have learned things from this book that you can incorporate into your day-to-day life while creating stunning software.

Solving performance problems usually comes down to a willingness to investigate things further. More often than not, it requires understanding the hardware and underlying OS well enough to be able to draw conclusions from measurement data. This book has scratched the surface in these areas when I've felt it necessary. After writing about C++20 features in this second edition, I'm now looking forward to starting to use these features in my profession as a software developer. As I've mentioned previously, a lot of the code presented in this book is only partially supported by the compilers today. I will keep updating the GitHub repository and adding information about compiler support. Best of luck!

Share your experience

Thank you for taking the time to read this book. If you enjoyed this book, help others to find it. Leave a review at **https://www.amazon.com/dp/1839216549**.