Chapter 5. Random Number Generation and Concurrency

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *learnmodcppfinance@gmail.com*.

Introduction

Of all the new features that were introduced in C++11, two that can have immediate and significant impact on very common tasks in quantitative financial development were random number generation and task-based concurrency. However, despite their usefulness, they have—for the most part—flown under the radar "in the literature". More specifically, random numbers can now be generated from a number of probability distributions provided in the Standard Library. Furthermore, the Standard Library also provides a cross-platform solution for implementing parallelizable tasks such as those that frequently occur in option pricing and risk management. This is thanks to an abstraction that takes care of all the thread management for us behind the scenes, much like a `vector` or a `unique_ptr` takes responsibility for memory management.

Beginning with C++17, and with significantly greater fanfare, the Standard Library began providing parallel versions of selected STL algorithms. What makes this particularly elegant is that as far as programmers are concerned, these algorithms can be executed in parallel by simply adding an extra argument at the call site. Note however that these arguments are requests, not constraints: a sequential execution of an algorithm is a valid parallel execution of that algorithm, so your library vendor can ignore such a request and remain conformant.

In this chapter, we will cover all three topics, beginning with random number generation and its applications in Monte Carlo option pricing. We will have a look at parallel STL algorithms, and then we will return to Monte Carlo models and see how implementing random equity price path scenarios as parallelizable tasks can significantly increase performance with minimal modifications to the code.

Distributional Random Number Generation

A very common task for anyone who has worked as a developer supporting an options trading desk or a risk management group is to generate random draws from a standard normal distribution. This typically required implementing a uniform random number generator using the Mersenne Twister **{1}** algorithm, and then applying a pairwise Box-Muller or polar rejection transformation **{2}**, resulting in two standard normal values. This of course required development time and extensive testing before it could go into production. It is also easy to make mistakes when implementing a random number generation algorithm, as it can be hard for humans to distinguish a sequence of random numbers from one that is actually predictable. Furthermore, quant developers changing jobs might find themselves repeating the very same task from scratch when starting positions at a new firm.

This process is now much easier and quicker to implement thanks to a library of random number engines and distributions introduced in the C++11 Standard. Both a uniform random number generator and a standard normal transformation algorithm are part of the ISO Standard, as are many other generators and distributions. In addition, as these utilities have been tested thoroughly by major Standard Library vendors, it obviates the need to devote a lot of resources to testing the generator algorithms themselves.

Engines and Distributions

An engine is an object that generates a sequence of large pseudorandom integer values based on an initializing unsigned integer seed. For each seed, a different sequence will be generated. If no seed is provided, a default seed is used; this default value can vary from engine to engine as well as among different library vendors. An engine object maintains state in the sense that it will return the next number in sequence when its function call operator -- taking no argument -- is called.

There are a variety of engines in the Standard Library. First is a default engine, `std::default_random_engine`. According to the ISO Standard, "on the basis of performance, size, quality, or any combination of such factors, . . . [the default engine is intended] for relatively casual, inexpert, and/or lightweight use" **{3}**. As the implementation is vendor-dependent, the actual numerical results are not guaranteed to be consistent across different Standard Library platforms. However, in the limit, each engine will approximate a random sequence of uniform integers for a given seed.

The Standard Library also provides several different pre-configured engine types based on well-known pseudorandom number generating algorithms, a full list of which can also be found in **{3}**. Templates are additionally provided that allow users to implement their own generators, but this is a more advanced topic beyond the scope of this book.

A distribution object depends on the state of an engine object. It is also a functor, but one that takes in the engine object as the argument. Each time the functor is called, the next value in the engine sequence is generated, and then the distribution object applies a transformation that provides a random draw from that particular distribution.

A distribution object is an instance of a class template, where the template argument is the numerical type being generated. There are default types for continuous distributions (type `double`) as well as for discrete distributions (type `int`).

As a first example, we can populate a vector with the first 10 random integers generated by the default engine with a seed of 100. The `<random>` header is required for for what follows.

```
#include <random>

. . .

using std::vector, std::cout;

// def is a functor, created with seed 100 and callable with no argument
std::default_random_engine def{ 100 };

for (int i = 0; i < 10; ++i)
{
      // Random integers generated directly from the engine, but
      // without specifying a distribution:
            cout << def() << " ";     //
      }
```

Using the Standard Library that accompanies the Visual Studio 2022 compiler, the results are:

```
(Output)
2333906440 2882591512 1195587395 1769725799 1823289175 2260795471 3628285872 638252938 20267358 673068980
```

As noted previously, default random engine implementations are vendor-dependent, as can be seen by the results from the Standard Library (libc++) that ships with the Clang compiler **{4}**:

```
(Output)
1568116515 3579517472 1937914647 3828374553 2130562958 4133907349 324794275 4144918260 2455697305 2309453985
```

In either case, these values alone are not terribly useful to us, but we can apply a distribution to transform them to random draws from, for example, a uniform distribution of real (floating point) numbers on the interval $[0, 1)$ (or a general half-open interval two real numbers $[a, b)$,

```
#include <random>;

std::vector<double> unifs(10);
std::default_random_engine def{ 30 };           // seed = 30
std::uniform_real_distribution<double> unif_rand_dist{ 0.0, 1.0 };

// Generate 10 uniform variates from [0.0, 1.0) and set
// as elements in the vector 'unifs':
for (double& x : unifs)
{
      x = unif_rand_dist(def);
}
```

The results in Visual Studio 2022 are then:

```
(Output)
0.916978 0.218257 0.409313 0.643062 0.706414 0.600071 0.927394 0.696651 0.0235988 0.440224
```

while with libc++ , we would get:

```
(Output)
0.946134 0.759504 0.568776 0.365942 0.0407869 0.582491 0.0371633 0.891006 0.230946 0.00526713
```

Each time the `unif_rand_dist` functor is called with the engine object as its argument, a distinct uniform random variate between zero and one is generated.

Note that if we ran this same code again inside the same scope, the results would be different, as the state of the `def` engine itself will be advanced to the next integer value. The sample code accompanying this chapter provides more examples.

As mentioned previously, for continuous random distributions, the default template parameter is `double`, so the uniform distribution object in this example could instead be defined as follows:

```
// Default template parameter is 'double'
std::uniform_real_distribution<> unif_rand_dist{ 0.0, 1.0 };
```

Remaining examples of continuous distributions will assume this default.

Generating Random Normal Draws

Simulations in quant finance are often based on Brownian motion **{5}**, in which a necessary step is to first generate a sequence of random draws from a standard normal distribution ($N(0, 1)$). The procedure is

essentially the same as in the previous example but with the uniform distribution replaced by the normal distribution.

There are tradeoffs between speed and quality, but of all the algorithms provided by the Standard, the Mersenne Twister is generally regarded as providing the best numerical results in financial applications, as it "has the longest non-repeating sequence with the most desirable spectral characteristics" **{6}**, although it is recommended "to have a small library of number generators available" **{6}**. That being said, we will primarily rely on the 64-bit Mersenne Twister engine, `std::mt19937_64`, in our examples. The `std::normal_distribution` class takes the mean and standard deviation as its constructor arguments, with defaults 0 and 1, so these are optional, as shown in this example:

```
std::vector<double> norms(10);
std::mt19937_64 mt{ 40 };
std::normal_distribution<> st_norm_rand_dist{};

// Generate 10 standard normal variates and set as elements in the vector 'norms':
for (double& x : norms)
{
      x = st_norm_rand_dist(mt);
}
```

Results in Visual Studio 2022 are:

```
(Output)
-0.872822 0.781329 0.302763 -1.13884 -0.0833526 0.336842 0.274821 1.83109 -0.4658 1.64151
```

While in Clang (libc++), we would get

```
(Output)
0.781329 -0.872822 -1.13884 0.302763 0.336842 -0.0833526 1.83109 0.274821 1.64151 -0.4658
```

For the general case, the mean and standard deviation are provided as constructor parameters, eg 0.25 and 0.15:

```
std::normal_distribution<> non_st_norm_dist{0.25, 0.15};
```

Other Distributions

There are two other distributions in `<random>` that can be useful in financial contexts. One is the Student's t-distribution, as it can be fit to the fatter tails (higher kurtosis) typical in financial returns data that are often not adequately captured with a normal distribution. Another example is the Poisson distri-

bution, often used for modeling the number of arrivals of tick data between two points in time, e.g. over each hour of the trading day.

The class for the t-distribution is the `std::student_t_distribution` class, and for the Poisson it is the `std::poisson_distribution` class. The same Mersenne Twister engine is used for both cases in the example below. The default `double` template parameter is assumed for the continuous t-distribution, and `int` for the discrete Poisson; hence, the template parameter for each can be left blank.

```
std::vector<double> t_draws(10);
std::vector<double> p_draws(10);

std::mt19937_64 mt{ 25 };
std::student_t_distribution<> stu{ 3 };
std::poisson_distribution<> pssn{ 7.5 };

for (auto& x : t_draws)
{
      x = stu(mt);
}

for (auto& x : p_draws)
{
      x = pssn(mt);
}
```

Unfortunately, there are no four-parameter distributions (as of yet) such as the Generalized Lambda or Generalized Hyperbolic distribution—common go-to distributions for fitting returns data—that can also fit the skewness and kurtosis of financial returns. A complete list of pre-defined distributions available in the Standard Library can be found on cppreference.com **{7}**.

NOTE

There are three additional options that can be adapted to a custom set of densities and intervals: `discrete_distribution`, `piecewise_constant_distribution`, and `piecewise_linear_distribution`. These are beyond the scope of this book, but more information can be found in **{8}** and **{3}**.

Shuffling

The `std::shuffle` algorithm, introduced in C++11, randomly rearranges the elements of a container with random access iterators using a random number generation engine as its source for randomness.

For example:

```
#include <algorithm>

// . . .

std:: mt19937_64 mt { 0 };
std::vector<int> v{1, 2, 3};
std::shuffle(v.begin(), v.end(), mt);
```

Each time this is called, it will generate a new order of the first three positive integers and store the results in the same vector `v`; thus, it is an order-modifying algorithm. Applying the `shuffle` algorithm as above three times and noting the results of each run, we might get something like the following:

```
(Output)
2 1 3
3 1 2
1 3 2
```

A range-based version was also introduced in C++20:

```
#include <ranges>

// . . .

// Ranges version of shuffle:
std::ranges::shuffle(v, mt);
```

A more interesting example comes out of systematic trading. Suppose you are backtesting an automated trading strategy that generates a sequence of P/L (Profit and Loss) values from round-trip trades. A common risk metric is a confidence level (eg 95%, 99%) for the worst possible maximum drawdown experienced by a series of automated trades.

Based on the example in Chapter Four of the popular book on trading strategies, *Trading Systems (2 Ed)*, by Jaekle and Tomasini **{9}**, the order of the P/L values is randomly rearranged multiple times without replacement, and the maximum drawdown is recorded for each run. The results could be fit to a statistical distribution, such as the normal distribution, or a non-parametric approach could also be used. The latter will be presented first in the example that follows.

To begin, suppose you have a strategy for which a backtest is run, and for simplicity let us assume the following 53 P/L values are the result of daily round trip trades (in general, this set of data could be much larger, and the trades could be booked in irregular intervals, ranging from non-uniform fractions of a second, or hours of a day, to varying numbers of days, or even months, depending on the strategy). The results in this example are loaded into a `vector` that we will call `pnl` (P/L):

```
vector pnl =
{
      -149'299.30,  -673'165.13,  3'891'123.79, 1'061'346.21,  -578'464.00,
      -260'855.99,       1'102'167.76,          509'764.96,       -276'786.46,          -11'947.13,
      -277'781.70,   -318'603.05,          57'747.06,    151'336.99,    267'826.14,
      -198'132.05,   -175'232.84,           -5'973.61,    177'224.51,    -711'878.76,
      -276'786.56,          116'489.15,   -185'188.30,       1'183'810.47,         1'563'147.38,
       -83'632.98,    527'687.09,         -307'650.96,       -321'589.76,       -1'434'711.00,
       742'743.96,   -245'921.36,         -198'131.26,   399'250.02,         -311'632.90,
       326'569.83,          437'084.67,        -297'694.80,       -379'336.81,          -173'240.27,
       -62'724.74,   -363'407.14,         -142'375.97,         -103'546.19,          187'179.27,
      -161'293.14,   -131'423.78,  1'195'759.19,          198'131.95,    -229'991.59,
      -109'519.00,         -148'348.69,       1'447'621.95
};
```

Before proceeding with determining a confidence level, we need to look at how to calculate drawdowns from a single backtest. This starts by generating the *cumulative* P/L over the backtest period. This will give us the resulting *equity curve*. This is easily obtained by appealing to the `std::partial_sum` algorithm first presented in Chapter Four. The results are appended to a new `vector`, say `cum_pl`:

```
std::vector<double> cum_pl;
std::partial_sum(pnl.begin(), pnl.end(), std::back_inserter(cum_pl));
```

Visually, this is represented in Figure 5-1:

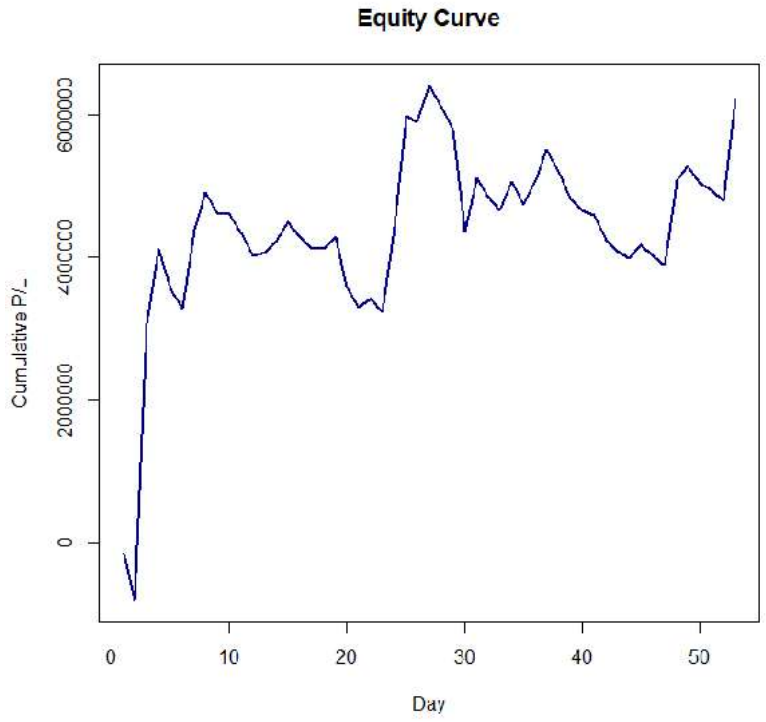

Figure 5-1: Equity Curve from Cumulative P/L Values

There are various definitions of drawdown, but for our purposes we will take it to mean the commonly-used difference between a given cumulative P/L value and its nearest previous peak value, as now shown in Figure 5-2, below the original equity curve:

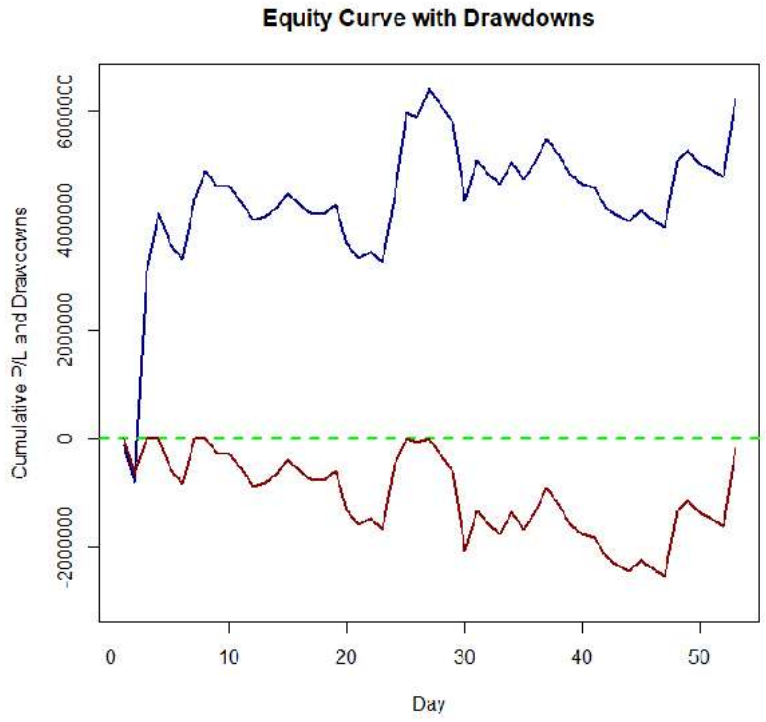

Figure 5-2: Equity Curve with Drawdowns

A `vector` called `drawdowns` can then be generated by iterating over `cum_pl`, identifying each new peak cumulative P/L value, measuring each drawdown, and appending the result with `push_back(.)`.

This is an instance where an iterator-based `for` loop can come in handy. It starts from the second position, as the `peak` value is initialized with the first P/L value so that a comparison can be performed at the outset of the loop:

```
double peak = cum_pl.front();
vector<double> drawdowns;
drawdowns.reserve(cum_pl.size());

for (auto pos = cum_pl.begin() + 1; pos != cum_pl.end(); ++pos)
{
    if (peak < *pos)
    {
        peak = *pos;
    }

    drawdowns.push_back(peak - *pos);
}
```

Having a `vector` of drawdowns is often desired for backtest analyses, but what we are mainly concerned with here is the maximum drawdown, which is easily obtained with our old friend the `max_element` algorithm:

```
double max_dd = *std::ranges::max_element(drawdowns);
```

Furthermore, we can get not only the maximum drawdown value over the backtest, but also the net profit (or loss, if negative) from the last term in the resulting `cum_pl` container. A backtest metric that is typically more useful than the net P/L value alone is the ratio $\frac{Net\;Profit}{Max\;DD}$, which is also easily obtained:

```
double net_pl_over_max_dd = cum_pl.back() / max_dd;
```

Displaying the output

```
cout << std::fixed << std::setprecision(2)
     << "Max DD from backtest = \$" << max_dd << ", Net P/L = \$" << cum_pl.back()
     << ", (Net P/L)/MaxDD = " << net_pl_over_max_dd << "\n;
```

gives us `Max DD from backtest` = \$2,541,852.33, `Net P/L` = \$6,237,745.13, and `(Net P/L)/MaxDD` = 2.45.

You may also often find drawdown expressed in percentage format, for which you could just substitute the following in for the last line of the `for` loop above:

```
drawdowns.push_back(1.0 - *pos/peak);
```

This yields 39.56%.

Results from a single backtest can provide useful preliminary information about the viability of a trading strategy, but to get a measure of the risk, Monte Carlo simulations can be run to get an empirical worst case maximum drawdown for a given level of confidence, using a method similar to that presented in **{9}**. The way this works is to rearrange the round trip P/L trade values a “large” number of times, generate the cumulative P/L values, and compute the maximum drawdown for each scenario.

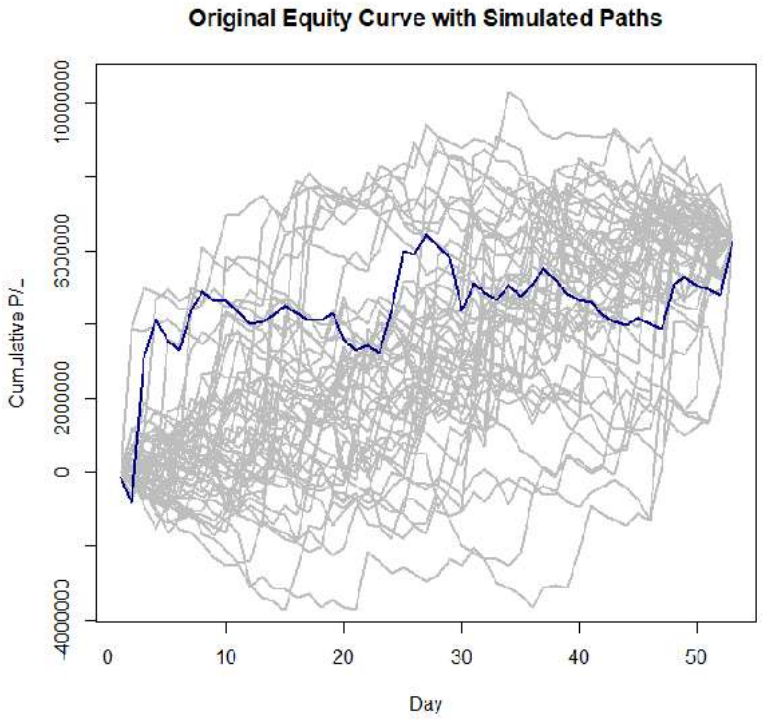

Figure 5-3: Original Equity Curve and Shuffled Simulations

The figure above (5-3) shows the original equity line from the backtest (in blue), and a set of (50) equity lines resulting from different permutations of the trade P/L values. The starting and ending points will be the same, since we are sampling *without replacement*. As noted in Jaekle and Tomasini, "[t]he change of the trade orders leads to the effect that between the [permuted] equity curves there are big variations with different drawdown phases that occur at different times." **{9}**.

In order to get a measure of the risk due to drawdown, we will apply the `shuffle` algorithm a "large" number of times to the `pnl` container, obtain the maximum drawdown on each run, and store each result in a new `vector`. This will give us a set of data from which we can infer the worst possible maximum drawdown for a given confidence level. As this process of permuting the P/L values is repeated, it will be easier to wrap the previous code into a reusable lambda expression, `max_dd_lam`, as shown in the code that follows, and return the maximum drawdown for each simulation. In this case, we will forgo generating a full `vector` of drawdown values for each run and just concentrate on obtaining the maximum drawdown value alone.

```
auto max_dd_lam = [](const vector<double>& v)
{
    std::vector<double> cum_pl;
    cum_pl.reserve(v.size());
    double max_dd = 0.0;
```

```
        std::partial_sum(v.begin(), v.end(), std::back_inserter(cum_pl));
        double peak = cum_pl.front();

        for (auto pos = cum_pl.cbegin() + 1; pos != cum_pl.cend(); ++pos)
        {
            if (*pos < peak)
            {
                max_dd = std::max(peak - *pos, max_dd);
            }
            else if (peak < *pos)
            {
                peak = *pos;
            }
            // else:  continue
        }
        return max_dd;
};
```

To demonstrate, set `n` to 100 (in actual trading applications, a larger number of samples will probably be required). We will also need a `vector` to store the maximum drawdown value from each simulation, say `max_drawdowns`:

```
unsigned n = 100;

vector<double> max_drawdowns;
max_drawdowns.reserve(n);
```

We can begin by appending the maximum drawdown from the original backtest order:

```
max_drawdowns.push_back(max_dd);
```

Next, define a 64-bit Mersenne Twister random engine with a seed of 10 to use in the `shuffle` algorithm:

```
std::mt19937_64 mt{ 10 };
```

Then, to generate a different random permutation of the `pnl` container `n - 1` times by shuffling it and then inputting it into the lambda to get the maximum drawdown for each run. Each result is also stored in the `max_drawdowns` container:

```
for (unsigned k = 0; k < n - 1; ++k)
{
    std::ranges::shuffle(pnl, mt);
```

```
        max_drawdowns.push_back(max_dd_lam(pnl));
    }
```

We write a lambda to round these values to two decimal places, and then use it as an auxiliary function in the range-based `sort` algorithm introduced in the previous chapter:

```
auto print_dec_form = [](double x) { cout << std::fixed << std::setprecision(2) << x << " "; };
std::ranges::for_each(max_drawdowns, print_dec_form);
```

The resulting values are as follows:

```
(Output)
 2541852.33 2285932.00 3511675.10 2346711.28 2320940.63 2081873.15 2726999.31 3517694.65 2800719.66 2862401.25 2392511.81 4295229.34 4417633.14 205
```

To obtain, say, a 95% confidence level, one approach is to fit the resulting data to a normal distribution. This is similar to the method described in **{9}**. Theoretically, this confidence level is

$$\mu + \sigma Z_{0.95}$$

where μ and σ are the mean and standard deviation of the fitted distribution, and $Z_{0.95}$ is the 95%-tile of the Standard Normal distribution, which equals approximately 1.64485.

Maximum likelihood estimators of μ and σ can be obtained by writing a lambda, `norm_params`, as follows, with the results stored in an STL `array`:

```
using std::array;

auto norm_params = [](const vector<double>& v)
{
    double mean = (1.0 / v.size()) * std::accumulate(v.begin(), v.end(), 0.0);

    double sum_sq = 0.0;
    for (double val : v)
    {
        sum_sq += (val - mean) * (val - mean);
    }

    return array<double, 2> { mean, (1.0 / std::sqrt(v.size()))* std::sqrt(sum_sq) };
};

double mean = norm_params(max_drawdowns)[0];
double sd = norm_params(max_drawdowns)[1];
```

If imposing the assumption of normality on the distribution of simulated maximum drawdowns, as is the case in **{9}**, this can be demonstrated for now by again setting $\alpha = 5%$. A simplifying assumption sometimes used in parametric risk calculations is to take the sample standard deviation as the "known" standard deviation of the distribution σ **{10'}**. The mean $\bar{X}$ and sample standard deviation s are obtained from the data using the lambda expression `norm_params` and calculating the usual upper confidence limit for "large" n:

$$\overline{X} + \sigma Z_{0.95}$$

where $Z_{0.95} = 1.64485$ is the 95th percentile from the standard normal distribution. Implementing this in code:

```
const double z_val = 1.64485;
double upper_conf_intvl_norm = mean + sd * z_val;
```

gives us 4,386,226.63, rounded to two decimal places.

An obvious question here is how would we allow for other levels of significance, say 97.5% or 99%? Although the Standard Library now provides random number generation from a normal distribution (among others as well), it does not provide the probability density function (pdf), cumulative distribution function (cdf), or the percentile function. So for now, the example has assumed the hard-coded 95th percentile value of 1.64485. The Mathematical Toolkit in the Boost Libraries, however, does provide these functions, and they will be covered in Chapter Eight.

Another method often employed in Monte Carlo risk modeling is to determine a non-parametric confidence level for the worst possible loss. The first step would be to sort the maximum drawdown values in ascending order, which will give us a chance to apply the range-based `sort` algorithm introduced in the previous chapter:

```
std::ranges::sort(max_drawdowns);
```

Again rounded to two decimal places, this gives us:

```
(Output)
1745348.76 1753314.05 1811060.31 1819025.50 1828980.34 1864824.72 1872790.60 1945470.30 1951446.10 1954544.56 1958408.76 1983258.00 2059969.17 20
```

As a naive approach to determining a measure of the 95% confidence level, we could simply eyeball the top five sorted maximum drawdown values comprising the top 5% (4,744,315.84 4,761,240.15 4,777,120.20 5,317,756.56 5,687,134.51), leaving 4,443,583.81 for the 95%-tile value.

In practice, of course, we would want to automate this, and the `std::lround(.)` function, introduced in `<cmath>` with C++11, will round to the nearest integer value and return it as a `long` type. This gives us

the position to which to iterate from the end of the `vector`, providing the same "worst case" maximum drawdown value in the preceding position.

```
// Assume alpha is taken in as user input:
double alpha = 0.05;          // Upper 5%-tile
long upper_loc = std::lround(alpha * max_drawdowns.size());          // Rounding function from C++11
double max_dd_conf_lev = *(max_drawdowns.end() - upper_loc - 1);
```

This would again give us 4,443,583.81.

NOTE

Prior to C++11 and the `shuffle` algorithm, the only built-in choice was `random_shuffle`. This was deprecated in C++14 and removed from the Standard Library with the release of C++17. This deprecation was due to its dependence on `std::rand(.)`, a C function with limited period that depends on global mutable state, making it dangerous to use in contemporary multithreaded situations. The fact that the `shuffle` algorithm takes an engine object as an argument removes this dependence on shared mutable state. Furthermore, the standard random number generation engines since C++11 have much longer periods than `std::rand(.)` did, which makes modern C++ utilities significantly better than their predecessors.

Monte Carlo Option Pricing

One of the most popular numerical approaches to pricing options, and in particular path-dependent options with European-type payoffs, also uses Monte Carlo simulation. Where it differs from the previous drawdown example is that it is based on the no-arbitrage pricing theory and Brownian motion, rather than an empirical method of rearranging trades.

For European vanilla options, the closed-form Black-Scholes solution can of course be used, but the illustration here can be used to extend the code to handle more complex path-dependent options, such as where barriers, ratchets, and lookbacks are involved.

One thing to note before we continue, as some readers may be aware, and as mentioned in Chapter Three, basic Monte Carlo methods as those that follow are not suitable for American options, as there is no accounting for optimal early exercise. An alternative, binomial lattice models, to be discussed in Chapter Eight, will handle options with early exercise.

A Review of Monte Carlo Option Pricing

For a European option, Monte Carlo simulations of randomly generated equity price paths projected to the expiration date are randomly generated using a discretized stochastic process that falls out of the Black-Scholes/no-arbitrage theory **{10}**. These simulated price paths are often referred to as *scenarios* in practice, and so to keep the terminology more compact, this is the term we will use.

The time to expiration from time $t = 0$ to some $t = T > 0$ is chopped into smaller and equal time intervals Δt, where

$$\Delta t = t_i - t_{i-1} i = 1, 2, \cdots, n$$

and

$$t_0 = 0 < t_1 < t_2 < \cdots < t_n = T$$

The equity price at time t is denoted S_t, and the strike price as X. The terminal price S_T is then compared with X to compute the payoff at expiration.

The stochastic process that governs the random equity prices is given by

$$S_t = S_{t-1} e^{r-q-\frac{\sigma^2}{2}+\sigma\varepsilon_t\sqrt{\Delta t}}$$

where r represents the annual continuous risk-free rate, q the annualized continuous dividend rate, σ the annualized volatility of the equity returns, and ε_t an independent standard normal random variate at each time t. The current spot price observed in the market is assigned to S_0 to initialize each random path.

The option price is then determined by taking the average of all the computed payoff values back to $t_0 = 0$.

As an example, suppose you are pricing a European call option with strike = \$105 on an equity paying no dividend that currently trades for \$100/share. Suppose further it is early September, and this option expires four months from now in January. Finally, assume the market indicates the annualized US Treasury Bill rate over this period is 1.2%. In a simplified case where five scenarios (random equity paths) are generated, the terminal prices might be \$120, \$115, \$110, \$100, and \$95. The first three result in in-the-money payoffs of \$15, \$10, and \$5 respectively, while the last two are out-of-the-money, in which cases the option would expire worthless.

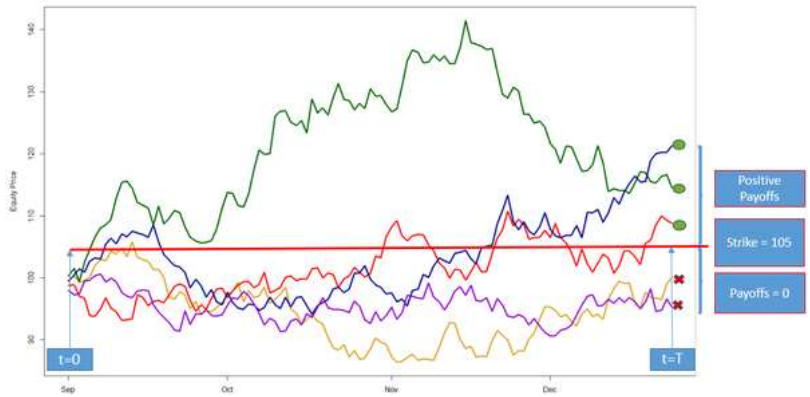

Figure 5-4: Simulated Equity Price Paths with Monte Carlo Methods

The Monte Carlo valuation of this option is then the mean of the payoffs discounted back by four months (one third of a year). Note that all payoffs must be accounted for, including the last two that expire worthless.

$$\frac{e^{-0.012)\frac{1}{3}(120-105)+(115-105)+(110-105)+0+0\}}}{5}$$

To get convergence to a meaningful value representing the price of the option, we of course would need to extend this to many more scenarios, usually requiring a range of at least 20,000 to maybe even 100,000 or more. Visually, this can start to resemble a Jimi Hendrix Experience album cover.

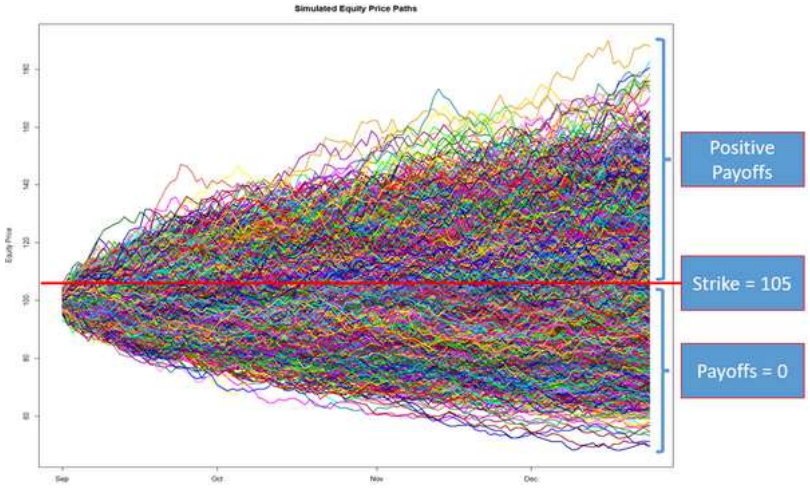

Figure 5-5: Tens of Thousands of Monte Carlo Scenarios

The next steps will describe how Monte Carlo option pricing can be implemented in modern C++.

Generating Random Equity Price Scenarios

Our procedure starts with generating a single equity price scenario. We will then repeat that process many times over to obtain a sufficient number of scenarios for calculating the option price. To start, we will first write a class called `EquityPriceGenerator`, which takes in as arguments at construction:

- The spot equity share price
- The number of equally-distanced time steps in one scenario
- The time (in years, or year fraction) left until option expiration
- The underlying stock volatility
- The risk-free interest rate
- The dividend rate (if any)

The spot price represents the initial price in a random scenario, S_0, as noted above. A functor is declared whose function call operator takes as its argument an `unsigned` seed value and will be responsible for generating the series of prices and returning them in a `vector`.

The class declaration could then be written as follows:

```
#include <vector>

class EquityPriceGenerator
{
public:
	EquityPriceGenerator(double spot, unsigned num_time_steps,
		double time_to_expiration, double volatility, double rf_rate, double div_rate);

	// Returns the simulated random path of equity share prices
	std::vector<double> operator()(unsigned seed) const;

private:
	double spot_;
	unsigned num_time_steps_;
	double time_to_expiration_;
	double volatility_;
	double rf_rate_;		// Continuous risk-free rate
	double div_rate_;		// Continuous dividend rate
	double dt_;			// dt_ = "delta t"
};
```

With the member variables initialized at construction, the operator implementation defining the functor is responsible for the random generation of standard normal variates, and the calculation of each simulated price. Note that the lambda expression implements equation (3-1). In order not to obfuscate the main point of this example, we will make the simplifying assumption that all of constructor arguments greater than zero.

```
#include "EquityPriceGenerator.h"
#include <cmath>
#include <random>
#include <algorithm>

EquityPriceGenerator::EquityPriceGenerator(double spot, unsigned num_time_steps,
	double time_to_expiration, double volatility, double rf_rate, double div_rate) :
	spot_{ spot }, num_time_steps_{ num_time_steps }, time_to_expiration_{ time_to_expiration },
	volatility_{ volatility }, rf_rate_{ rf_rate }, div_rate_{ div_rate },
	dt_{ time_to_expiration / num_time_steps }{}

std::vector<double> EquityPriceGenerator::operator()(unsigned seed) const
{
	std::vector<double> v;
	v.reserve(num_time_steps_ + 1);

	std::mt19937_64 mt(seed);
	std::normal_distribution<> nd;
```

```
        auto new_price = [this](double previous_equity_price, double norm)
        {
            double price = 0.0;

            double exp_arg1 = (rf_rate_ - div_rate_ -
                ((volatility_ * volatility_) / 2.0)) * dt_;
            double exp_arg2 = volatility_ * norm * sqrt(dt_);
            price = previous_equity_price * std::exp(exp_arg1 + exp_arg2);

            return price;
        };

        v.push_back(spot_);           // put initial equity price into the 1st position in the vector
        double equity_price = spot_;

        for (unsigned i = 1; i <= num_time_steps_; ++i) // i <= num_time_steps_ since we need
                                                        // a price at the end of the final time step.
            equity_price = new_price(equity_price, nd(mt));     // norm = nd(mt)
            v.push_back(equity_price);
        }

        return v;
    }
```

Each of the simulated underlying equity prices is pushed onto the vector `v`, which is then returned. `v` will thus hold an individual scenario.

Calculating the Option Price

The above takes care of generating a single random price scenario. To calculate the option price, we will need to iterate through a set of seeds to generate thousands of distinct and varying scenarios, calculate the terminal payoffs, and then take the mean of the discounted payoffs. We can implement a class that takes in an `OptionInfo` object (as introduced in Chapter Three) with payoff set at runtime, along with market data and parameters needed for the equity price generator. If there is time left until expiration, a Monte Carlo simulation is performed by generating a series of random scenarios and calculating the average of the discounted payoffs. If not, then the value will be determined solely by the payoff at expiration, yielding the intrinsic value.

The declaration of the class could be written as follows. Recall that as an `OptionInfo` object (`opt`) will have unique pointer member pointing to its payoff type, it is taken in as an rvalue reference constructor argument. Recall also that `opt` will hold the time to expiration as member data.

For convenience, the dividend rate is defaulted to zero.

```
#include "OptionInfo.h"

class MCOptionValuation
{
public:
	MCOptionValuation(OptionInfo&& opt, unsigned time_steps, double vol,
		double int_rate, double div_rate = 0.0);

	double calc_price(double spot, unsigned num_scenarios, unsigned unif_start_seed);

private:
	OptionInfo opt_;
	unsigned time_steps_;
	double vol_, int_rate_, div_rate_;
};
```

In the source file, the constructor will again just be responsible for initializing the member data, and the option valuation will be performed by the `calc_price(.)` member function:

```
#include "MCOptionValuation.h"
#include "EquityPriceGenerator.h"

#include <random>
#include <utility>		// std::move
#include <cmath>
#include <vector>
#include <numeric>		// std::accumulate

MCOptionValuation::MCOptionValuation(OptionInfo&& opt, unsigned time_steps, double vol,
	double int_rate, double div_rate): opt_{ std::move(opt) }, time_steps_{ time_steps },
	vol_{ vol }, int_rate_{ int_rate }, div_rate_{ div_rate } {}

double MCOptionValuation::calc_price(double spot, unsigned num_scenarios, unsigned unif_start_seed)
{
	if (opt_.time_to_expiration() > 0.0)		// (1)
	{
		using std::vector;

		std::mt19937_64 mt_unif{unif_start_seed};
		std::uniform_int_distribution<unsigned> unif_int_dist{};	// (2)

		vector<double> discounted_payoffs;				// (3)
		discounted_payoffs.reserve(num_scenarios);
		const double disc_factor = std::exp(-int_rate_ * opt_.time_to_expiration());

		for (unsigned i = 0; i < num_scenarios; ++i)		// (4)
		{
			EquityPriceGenerator epg{ spot, time_steps_, opt_.time_to_expiration(), vol_,
```

```
                    int_rate_, div_rate_ };                         // (5)
                vector scenario = epg(unif_int_dist(mt_unif));  // (unif_int_dist(mt_unif): next seed)

                discounted_payoffs.push_back(disc_factor
                    * opt_.option_payoff(scenario.back()));    // (6)
            }

            return (1.0 / num_scenarios) * std::accumulate(discounted_payoffs.cbegin(),
                discounted_payoffs.cend(), 0.0);                // (7)
        }
        else
        {
            return opt_.option_payoff(spot);                    // (8)
        }
    }
```

In the `calc_price(.)` member function, it first checks whether there is time left to expiration (1), in which case Monte Carlo equity price scenarios will be generated. If this is the case, it creates a 64-bit Mersenne Twister engine that will drive a uniform integer distribution (2) over the interval of the minimum possible `unsigned` value to the maximum value. The purpose of this distribution is to generate random unsigned integer seed values to use as the seed argument for each `EquityPriceGenerator` instance generating a distinct scenario. The uniform integer distribution itself will require a seed for its Mersenne Twister engine, which is set by user input to the `unif_start_seed` function argument. This is one possible way to generate random seed values, and it does guarantee reproducibility of the individual price paths, which is sometimes required by trading desks and risk management departments.

Recalling that we will need to sum the discounted payoff values in order to compute the average to get the option value, a `vector` called `discounted_payoffs` is instantiated (3). And to prevent memory reallocation, its capacity is set to the number of scenarios from which each payoff will be obtained (using `reserve(.)`, as discussed in Chapter Four). The discount factor from expiration back to the valuation date ($t = 0$) is also calculated here and set to the constant `disc_factor`, to be applied to each simulated payoff.

Next, a `for` loop will run an iteration (4) for each scenario (eg 25,000 times). And for each time through, a new `EquityPriceGenerator` object will be created (5) using the next seed from the uniform integer distribution established at the outset. For a European option we only need the final price from each scenario, with which we can get the payoff at expiration by using it as the argument for the `option_payoff(.)` function on the same `opt_` member, thus avoiding the performance penalty of generating multiple `OptionInfo` objects.

The option price is then the mean of the discounted payoffs, and is returned. If there is no time remaining until expiration, then the `else` condition is executed, just returning the non-discounted payoff of the `OptionInfo` member (8).

As an example, suppose we have the following contract and market data (again, where in practice this information would come through an interface and would not be hard-coded):

```
double strike = 75.0;
double spot = 100.0;
double vol = 0.25;
double rate = 0.05;
double div = 0.075;
double time_to_exp = 0.5;
```

Suppose also we set the number of time steps for each of the 25,000 random equity price scenarios to ten, and that we set the seed for the uniform integer random generator to 42:

```
unsigned num_time_steps = 12;
unsigned num_scenarios = 25'000;
int seed = 42;
```

To price a call option, the first step is to create an `OptionInfo` object (reusing the concluding post-C++11 version of the class developed in Chapter Three), with a unique pointer to a `CallPayoff` object as a constructor argument. Next, the resulting option object is used to construct a `MCOptionValuation` object, along with the additional constructor arguments:

```
OptionInfo opt_call_itm_not_exp_with_div{ std::move(std::make_unique<CallPayoff>(strike)), time_to_exp };
MCOptionValuation val_call_itm_not_exp_with_div{ std::move(opt_call_itm_not_exp_with_div),
    num_time_steps, vol, rate, div };
```

The option value is then easily obtained by calling the `calc_price(.)` member function on the `MCOptionValuation` object, which takes in the spot price of the equity share price, along with the desired number of scenarios, and the seed for the uniform integer random generator which itself provides a random seed value for each scenario.

```
double opt_value = val_call_itm_not_exp_with_div.calc_price(spot, seed, num_scenarios);
```

The computed option value in this case is \$23.55. For comparison, the closed-form Black-Scholes price is \$23.52, so in this case convergence is very close.

Path-Dependent Options

The previous example has hopefully demonstrated the mechanics behind generating Monte Carlo equity price path scenarios and discounting the resulting payoffs to get a value of an option. However, it is not

all that practically useful, as we could just implement the closed-form Black Scholes formula to get the same result. Where Monte Carlo methods become more useful is in valuing *path-dependent* options.

For an example of path-dependent options, we can consider barrier options, specifically up-and-out and down-and-out cases. Closed-form solutions for single barrier option valuation that derive from Black Scholes theory also exist **{11}**, so we will have a benchmark against to test our results (these can be verified using tools such as **{12}** and **{13}**). The methods we are covering for both barrier and non-barrier options, however, can be extended to many types of options with more complex and/or multiple path-dependent payoffs, and for which closed-form solutions do not exist. Much of the typical machinery for generating the underlying scenarios is the same, or similar.

As for the valuation of barrier options, we can start by defining a scoped enumeration (see `enum class`, introduced in Chapter Two) to represent the three distinct possible barrier conditions:

```
enum class BarrierType
{
    none,
    up_and_out,
    down_and_out
};
```

The `MCOptionValuation` constructor can then be modified to accommodate different barrier types, with the defaults set to `BarrierType::none` and a barrier value of zero. A new member variable for each is also declared:

```
public:
    MCOptionValuation(OptionInfo&& opt, unsigned time_steps, double vol,
        double int_rate, double div_rate = 0.0, BarrierType barrier_type = BarrierType::none,
        double barrier_value = 0.0);

    // . . .

private:
    // . . .
    BarrierType barrier_type_;
    double barrier_value_;
```

The constructor implementation is again responsible for member variable initializations:

```
MCOptionValuation::MCOptionValuation(OptionInfo&& opt, unsigned time_steps, double vol,
    double int_rate, double div_rate, BarrierType barrier_type, double barrier_value) :
    opt_{ std::move(opt) }, time_steps_{ time_steps },
    vol_{ vol }, int_rate_{ int_rate }, div_rate_{ div_rate },
    barrier_type_{ barrier_type }, barrier_value_{ barrier_value } {}
```

The `calc_price(.)` function can be retrofitted to accommodate a barrier condition, as shown in the updated implementation that follows.

```
double MCOptionValuation::calc_price(double spot, unsigned num_scenarios, unsigned unif_start_seed)
{
    bool barrier_hit =
        (barrier_type_ == BarrierType::up_and_out && spot >= barrier_value_) ||
        (barrier_type_ == BarrierType::down_and_out && spot <= barrier_value_);        // (1)

    if (barrier_hit) return 0.0;   // Option is worthless        // (2)

    // Case where barrier has not (yet) been crossed

    if (opt_.time_to_expiration() > 0 )                                    // (3)
    {
        std::mt19937_64 mt_unif{ unif_start_seed };                        // (4)
        std::uniform_int_distribution<unsigned> unif_int_dist{};
        const double disc_factor = std::exp(-int_rate_ * opt_.time_to_expiration());

        using std::vector;
        vector<double> discounted_payoffs;                                 // (5)
        discounted_payoffs.reserve(num_scenarios);

        // Iteration starts with barrier_hit = false
        for (unsigned i = 0; i < num_scenarios; ++i)                 // (6)
        {
            EquityPriceGenerator epg{ spot, time_steps_, opt_.time_to_expiration(), vol_,
                int_rate_, div_rate_ };
            vector scenario = epg(unif_int_dist(mt_unif));         // (7)

            switch (barrier_type_)                                         // (8)
            {
                case BarrierType::none: break;                             // (9)

                case BarrierType::up_and_out:                              // (10)
                {
                    auto barrier_hit_pos = std::find_if(scenario.cbegin(), scenario.cend(),
                        [this](double sim_eq) { return sim_eq >= barrier_value_; });   // (11)
                    if (barrier_hit_pos != scenario.cend()) barrier_hit = true;        // (12)
                }
                break;

                case BarrierType::down_and_out:                            // (13)
                {
                    auto barrier_hit_pos = std::ranges::find_if(scenario,
                        [this](double sim_eq) { return sim_eq <= barrier_value_; });   // (14)
                    if (barrier_hit_pos != scenario.cend()) barrier_hit = true;
                }
                break;
```

```
                }     // end of switch statement

                if (barrier_hit)
                {
                    discounted_payoffs.push_back(0.0);                    // (15)
                }
                else
                {
                    discounted_payoffs.push_back(disc_factor * opt_.option_payoff(scenario.back())); // (16)
                }

                barrier_hit = false;
            }

            // Option value = mean of discounted payoffs
            return (1.0 / num_scenarios) * std::accumulate(discounted_payoffs.cbegin(),
                discounted_payoffs.cend(), 0.0);                          // (17)
        }
        else                                                              // (18)
        {
            // At expiration, barrier_hit == false
            return opt_.option_payoff(spot);                              // (19)
        }
    }
```

In this implementation, with the presence of barriers, there is of course more work to do. To start, we need to check if the underlying equity has already crossed the barrier (1). If so, it is worthless, so a value of zero is returned (2).

Next, similar to the previous non-barrier example, the interesting cases will be where there is positive time left to expiration ($t > 0$) (3). We start by setting up a random sequence of integer seed values, and defining the discount factor (`disc_factor`) to be applied throughout each scenario generation (4). The `discounted_payoffs` container, which will again hold the discounted payoff from each scenario, is also created, with memory reserved to avoid memory reallocation of the `vector` storage (5).

Inside the loop (6), with each iteration, we will again generate a distinct random equity price scenario based on a random seed value drawn from a uniform integer distribution (7). Now, a `switch` / `case` statement is set up based on each `BarrierType` enumerator (8). In the event there is no barrier, so the option is a plain European option with $t > 0$, which will have a terminal payoff (9).

If an up-and-out barrier is present (10), the code will need to iterate through the generated scenario to determine whether the barrier was crossed. Note this can be accomplished more elegantly than a `for` loop by using the `find_if` algorithm (11). This will return the position of the first simulated price in the `scenario` container that exceeds the barrier, if it exists, which is conditioned on whether the iterator

`barrier_hit_pos` reaches the `end` position or not. If so, `barrier_hit` is now set to `true` (12). If not, similar to the plain European case, there will be a terminal payoff.

The logic is similar for a down-and-out barrier (13), but note we can write even more modern and cleaner code by using the ranges version of `find_if` (14). Whether the iterator form or ranges form, using the `find_if` algorithm will be preferable to embedding a messier and more error-prone nested multiline `for` loop such as

```
// Don't write this -- replace with find_if algorithm
// . . .
for (double sim_value : scenario)
{
      if (sim_value >= barrier_value_)
      {
            barrier_hit = true;
            break;          // break out of for loop
      }
}
```

This iteration ends with the discounted payoff being conditioned on whether the barrier was crossed, that is, whether `barrier_hit` is `true` or not. If so, with a knockout barrier, the option expires worthless (15), so the discounted payoff is zero. Otherwise, the payoff is treated as if it were from a plain European option (16), where the terminal payoff is discounted back to the present. In either case, the result is appended to the `discounted_payoffs` vector.

`barrier_hit` is reset to `false`, and then the loop repeats itself with a new seed provided by the random uniform integer generator in line (7). The process continues until the total number of discounted payoffs have been computed, and then the option value is calculated by again taking the mean, in line (17).

Finally, we need to cover the other trivial case where the option is being valued at expiration, in the concluding `else` block (18). We know that `barrier_hit` is `false` here, so the value of the option is just the European payoff (19).

NOTE

You might notice the `find_if` algorithm is placed inside braces in each of the barrier cases in the `switch` / `case` statement. The reason for the braces is that variables are being declared local to these scopes. Without the braces, you will end up with a compiler error.

As an example, we can revisit the previous non-barrier European call option example where we had

```
double strike = 75.0;
double spot = 100.0;
```

```
double vol = 0.25;
double rate = 0.05;
double div = 0.075;
double time_to_exp = 0.5;
unsigned num_time_steps = 12;
unsigned num_scenarios = 20'000;
int seed = 42;
```

We can then add an up-and-out barrier value of \$130.00:

```
BarrierType barr_type = BarrierType::up_and_out;
double barr_val = 130.0;
```

Now, setting this up, we again create an `OptionInfo` object, and pass it by move semantics to the `MCOptionValuation` constructor:

```
OptionInfo opt_call_itm_barr{ std::make_unique<CallPayoff>(strike), time_to_exp };
MCOptionValuation val_call_itm_barr{ std::move(opt_call_itm_barr),
      num_time_steps, vol, rate, div, barr_type, barr_val };
```

The option value is then obtained using the same member function, `calc_price(.)`:

```
double opt_val = val_call_itm_barr.calc_price(spot, seed, num_scenarios)
```

The result is \$19.61, which is less than the value of \$24.55 that we got in the equivalent non-barrier case. This is expected, with more scenarios resulting in payoffs of zero. However, in this case, convergence to the analytic solution of \$17.82 is not as close. We can improve the result by increasing the number of scenarios and time steps:

```
num_time_steps = 24'000;
num_scenarios = 50'000;
```

In this case, the calculated option price is \$17.97, a significant improvement in convergence.

Concluding Remarks (for now)

At this point, you might very well be thinking that instead of generating each scenario serially, it might be much more efficient instead to farm these out to individual processes and compute them in parallel. Being a prime example of an *embarrassingly parallelizable* algorithm, we will soon see how Monte Carlo simulations can be implemented within the context of task-based concurrency introduced with C++11. These relatively recent Standard Library additions—quality random number generation and task-based

concurrency—can play very nicely together to significantly boost computational efficiency, with minimal code modification.

Concurrency and Parallelism

Two very powerful new features that enable running parallel tasks have been added to the Standard Library in recent years. The first of which is parallel algorithms, released in C++17. The second, which we will call task-based concurrency, stems from the introduction in C++11 of the `std::async` function, and `std::future` objects. What is remarkable is that in both cases you can easily realize significant performance gains, particularly in terms of throughput, without the pain of writing lines of complex code responsible for creating thread pools, implementing parallel execution, and deactivating the threads. These mechanisms, being part of the Standard Library, provide a big advantage being platform-independent.

With modern C++, parallel algorithms can now be called by simply adding an extra argument with respect to the non-parallel counterpart, and writing concurrent code can sometimes be accomplished by just writing a few lines of extra code that would otherwise execute sequentially. The benefits you can reap overwhelm the minimal extra work required. Of course, things are not always that simple with parallel or concurrent code, but that is not specific to C++.

We will first present standard parallel algorithms, and then return to the Monte Carlo option pricing example to demonstrate task-based concurrency. As the latter is an embarrassingly parallelizable procedure, it can also be accomplished without having to worry about deadlocks and race conditions that can occur when writing parallel code.

Since most standard consumer laptop and desktop machines today allow multicore processing, the features presented here will almost surely enable your code to run in parallel if you use them. Note, however, that sequential code is a special case of parallel code, and that asking for an algorithm to be executed in parallel does not guarantee that you will get a non-sequential execution. Ultimately, your standard library vendor may decide to honor your request or not depending on various factors. For example, if the amount of data to process does not seem sufficient to compensate for the overhead associated with the request, the implementation could determine that a sequential execution will be faster.

On the flip side, it is possible your request for parallel execution might still be honored, but it will not necessarily guarantee better performance. In fact it will quite possibly be worse than sequential execution performance, again particularly in cases of smaller data sets. Examples of this will be shown shortly.

Parallel Algorithms from the Standard Library

If there were ever an epitome of the term leverage, it would apply to parallel algorithms from the Standard Library. Most STL algorithms, as of C++17, have overloads with an extra argument, its execution policy, that instructs it to run in parallel. It couldn't be easier.

When requesting an algorithm to be executed in parallel, the execution policy argument appears first, followed by the other arguments usually passed to that algorithm. The `<execution>` header needs to be included when utilizing execution policies.

Other than the possibility to provide an execution policy, parallel standard algorithms look like their "conventional" counterparts: they can be parameterized with a thread-safe (preferably) auxiliary function (often a predicate), and they can be either non-modifying or modifying, the latter case encompassing order modification, element modification, or possibly both. We will look at examples of each, including some mathematical examples.

Descriptive Statistics

Suppose you wish to analyze typical descriptive statistcs, such as maximum and minimum, and mean values, from a set of data.

To simulate a set of data, we can generate a sample of normal random variates and place the results in a `vector`. Then, we can run STL algorithms in parallel to find the maximum and minimum element, and after this calculate the mean of these random values. To run each of these in parallel, all that is needed is to place the parallel execution policy, `std::execution::par`, in the first argument position when calling the algorithms involved.

For generating the standard normal data to be used in the example, the usual Mersenne Twister and normal distribution objects are created, and then the random number generation is wrapped in a lambda. This lambda then serves as the auxiliary function in the `std::generate` algorithm, and `n =` 50,000 random variates are drawn. This algorithm is run sequentially because it mutates the state of the engine (`mt`) and distribution (`nd`), but at this stage we are only obtaining the data to be used in the parallel examples to follow.

```
#include <vector>
#include <algorithm>
#include <numeric>
#include <random>
#include <execution>

. . .

// Generate large number of normal variates:
std::mt19937_64 mt(25);
std::normal_distribution<> nd;

auto next_norm = [&mt, &nd]()
{
      return nd(mt);
};

const unsigned n = 500;
```

```
std::vector<double> norms(n);
std::generate(norms.begin(), norms.end(), next_norm);
```

The algorithms to search for and locate the maximum and minimum values can now be set up for parallel execution by including the parallel execution policy parameter in the first argument position of the `std::max_element` and `std::max_element` algorithms, as shown next. These overloaded algorithms again respectively return an iterator to the maximum and the minimum and values in the sequence:

```
auto max_norm = std::max_element(std::execution::par,
   par_norms.begin(), par_norms.end());
auto min_norm = std::min_element(std::execution::par,
   par_norms.begin(), par_norms.end());
```

To compute the mean, in previous examples we appealed to the `std::accumulate` algorithm to sum up the values in the vector first; however, this is one case where there is no parallel version of the algorithm. Not to despair, a new algorithm, `std::reduce` will perform this task in parallel by default:

```
double mean_parallel = 1.0 / par_norms.size()
      * std::reduce(par_norms.begin(), par_norms.end(), 0.0);
```

If desired, however, the execution policy may be explicitly indicated, as there are other options besides `std::par` (to be discussed in the conclusion of this section):

```
double mean_parallel = 1.0 / par_norms.size()
      * std::reduce(std::execution::par, par_norms.begin(), par_norms.end(), 0.0);
```

Inner Product

There is one more numeric algorithm, `std::inner_product`, that does not carry a parallel version. Instead, we can use another new (C++17) algorithm, `std::transform_reduce`, whose default behavior gives us the inner product of two containers. Note that `transform_reduce` is parallel by default, so similar to `std::reduce`, the `std::par` execution policy can be omitted:

```
std::vector<int> v{ 4, 5, 6 };
std::vector<int> w{ 1, 2, 3 };

double dot_prod = std::transform_reduce(v.begin(), v.end(), w.begin(), 0.0);
```

What this does is first calculate each of the element-by-element products, and then sums them in parallel. In other words, it is shorthand for

```
dot_prod = std::transform_reduce(v.begin(), v.end(), dq.begin(), 0,
    std::plus<int>{}, std::multiplies<int>{});
```

`transform_reduce` also admits the more general form where other operations can be used, such as taking the sum of differences between two vectors:

```
double sum_diff = std::transform_reduce(v.begin(), v.end(), w.begin(), 0.0,
    std::plus<double>{}, std::minus<double>{});
```

Note that the first operation—in this example the element-by-element subtraction to compute the differences—is the second operation argument (`std::minus`), while the second operation—here the sum of the differences (`std::plus`)—is the first argument.

The algorithm can also be applied to a single `vector` , such as computing its sum of squares:

```
double ssq = std::transform_reduce(v.begin(), v.end(), v.begin(), 0.0);
```

Performance and Guidance

The previous examples were kept very simple with small vector sizes in order to focus on the essentials of parallel STL algorithms; in practice, with small amounts of data like this, it is possible these algorithms will execute sequentially regardless of the execution policy used as argument, or conversely exhibit worse performance if the parallel request is honored. The motivation for parallel algorithms, of course, is to achieve faster execution when computing over larger amounts of data, so here we will look at an example to illustrate this.

Suppose we have a large random sample from a standard normal distribution, and for each value in the sample, we want to compute the approximation of the exponential function using the power series expansion

$e^x \approx \sum\limits_{k=0}^{n}\frac{x^k}{k!}$

for n “sufficiently large”.

For this, we will first use the `std::transform` algorithm in its default, sequential mode, and then compare it with the same algorithm but with a parallel execution policy. The process will be repeated for increasing values of n in order to compare the execution times of both approaches.

In the code example that follows, assume vectors `u` and `v` have been populated with identical sets of standard normal random variables, generated using the 64-bit Mersenne Twister engine and normal distribution as before. `num_elements` will hold the number of elements in each of `u` and `v` :

```
std::mt19937_64 mtre{ 100 };
std::normal_distribution<long double> nd;
std::vector<long double> v(num_elements);

auto next_norm = [&mtre, &nd](double x)
{
    return nd(mtre);
};

std::transform(v.begin(), v.end(), v.begin(), next_norm);
auto u = v;
```

Then, setting the first two terms in the power series to 1 and x, e^x can be approximated with the following lambda expression, where `terms` represents n, the number of terms in the summation:

```
auto exp_series = [&terms](double x) {
    double num = x;          // A standard normal variate
    double den = 1.0;
    double res = 1.0 + x;
    for (unsigned k = 2; k < terms; ++k)
    {
        num *= x;
        den *= static_cast<double>(k);
        res += num / den;
    }
    return res;
};
```

Running the `transform` algorithm first sequentially, followed by its overload using the parallel execution policy parameter, and replacing each element of the respective `vector` (`u` or `v`) with its approximate exponential value, our code can be written as:

```
std::transform(u.begin(), u.end(), u.begin(), exp_series);
std::transform(std::execution::par, v.begin(), v.end(), v.begin(), exp_series);
```

In tests again performed on an 8-core machine, with increasing values of n (`terms`) and `vector` sizes for `u` and `v`, the results were as follows:

Table 5-1. Exponential Approximation Performance Results

Number of vector Elements	Number of Terms in Sum	No Exec Policy (ms)	With Parallel Exec (ms)	Speedup
100	10	0.0014	0.0824	-98.30%
500	50	0.0403	0.1261	-68.04%
500,000	200	210.3203	38.9450	440.04%
1,000,000	200	414.8995	97.6039	325.08%
5,000,000	200	2095.3706	433.6819	383.16%
10,000,000	20,000	4144.3402	561.8325	637.65%

(Add graph later)

Like any powerful tool, you will need to know when it is appropriate to use parallel algorithms and when it is preferable to refrain from so doing **{14}**. As can be seen in these results, blindly applying a parallel execution policy to a standard algorithm might not improve performance by any significant degree, and in some cases, performance may even degrade, as can be seen in the first two runs with smaller `vector` sizes and lower numbers of terms in the power series. But after cranking these values up higher, the performance gains become more significant, yet the *only difference in the code was simply adding the parallel execution policy argument*.

Even more notable results can be obtained with a larger number of processors, such as those on a 20-core virtual server, presented in **{15}**.

Execution Policies: In General

In the above examples, with an existing pre-C++17 standard algorithm, parallel processing is enabled by adding the parallel execution policy as the first argument. Examples:

```
std::min_element(std::execution::par, par_norms.begin(),
   par_norms.end());

std::transform(std::execution::par, v.begin(), v.end(), v.begin(), exp_series);
```

When using an algorithm introduced in C++17 designed specifically for parallel processing, however, the parallel execution policy is implied and hence not required, as was shown in two previous examples we covered, namely `std::reduce` and `std::transform_reduce`:

```
std::reduce(par_norms.begin(), par_norms.end(), 0.0);
double dot_prod = std::transform_reduce(v.begin(), v.end(), w.begin(), 0.0);
```

Execution policies can still be specified explicitly in these cases if desired, however:

```
std::reduce(std::execution::par, par_norms.begin(), par_norms.end(), 0.0);
dot_prod = std::transform_reduce(std::execution::par, v.begin(), v.end(), w.begin(), 0.0);
```

Officially, and more generally, three execution policies were introduced in C++17:

- Sequential execution policy, `std::execution::seq` : Ensures that the algorithm may not be run parallel
- Parallel execution policy, `std::execution::par` : Requests that the algorithm is executed in parallel
- Parallel unsequenced policy, `std::execution::par_unseq` : Indicates that an algorithm may be executed in parallel and vectorized

Technically speaking, the sequential policy might not be exactly the same as when no execution policy is indicated (for a pre-C++17 algorithm), but it is similar. Also, the term *vectorized* refers to executions on platforms such as SIMD (Single Instruction Multiple Data), GPU (Graphics Processing Unit), etc.

For this book, we will be primarily interested in the implied versions of `reduce` and `transform_reduce` as described just above. Further details on execution policies can be found in more advanced texts focused on parallel computation. The NVIDIA website also contains examples and information on the vectorizing execution policy **{16}**.

NOTE

For completeness, a fourth execution policy was added to C++20: `std::execution::unseq`. This is related to vectorized operations executed in parallel on each individual thread and (again) is beyond the scope of this book.

Task-Based Concurrency

Before C++11, the C++ standard was silent on the question of multithreading, and it left considerations to that effect under the purview of the operating system and third-party libraries. During the first decade of the current century, articles such as **{17}** made it clear that this was not sufficient, and that the language needed rules to take multithreading into account in order to produce correct programs.

Since C++11, thus, the C++ language has acknowledged concurrency and parallelism, providing a sound memory model to clarify what a compiler could and could not do, and it has provided tools to help programmers address concurrency-related problems as well as write code destined for parallel execution.

There are many definitions for concurrency and parallelism, and the C++ standard itself does not provide a formal definition for either. What it does define are rules that help model such things as what a thread of execution is and how execution progresses in the presence of multiple threads. For the purpose of this book, what matters the most is that there is now built-in support in the Standard Library to enable multithreading on personal computers and servers with multicore processors.

Programming multithreaded code is notoriously more difficult than single-threaded code, particularly in the presence of mutable state, which requires one to take into account synchronization in order to avoid *data races*. Informally, a data race occurs when a given object (including such things as `int` and `bool` objects) is accessed concurrently by at least two threads, with at least one thread modifying the object, all this without synchronization. If a program contains a data race, it is essentially broken.

As part of the C++11 concurrency library, the spawning, management, and deactivating of threads can be abstracted away by using what we will refer to as *task-based concurrency*. This way, a programmer can concentrate on enabling the actual tasks of an application to run in parallel, rather than having to be concerned about the intricacies and complexity of thread programming. Furthermore, as this is part of the Standard Library, code utilizing task-based concurrency will compile and run cross-platform.

Much like a `vector` is an abstraction of a dynamic array that manages the allocation and deallocation of heap memory, a `std::future` object abstracts away the responsibilities associated with thread construction and startup necessary to execute a specific task asynchronously. It will even abstract away whether an actual thread of execution needs to be started on demand or if that thread is taken from a pool of threads that are already running. This can make writing parallelized code significantly simpler, leading to code that tends to be cleaner, less error-prone, and overall, less time-consuming to maintain. This is especially convenient for financial programming, where embarrassingly parallelizable models involving tasks that are independent of each other are common, such as when employing Monte Carlo methods.

Creating and Managing a Task

A concurrent task is set in motion by constructing a `std::future` object that handles the relevant function and its arguments. This object is created with the `std::async(.)` function. Both are defined in the Standard Library `<future>` header.

For example, suppose we have a function that squares two integers:

```
int square_val(int x)
{
   return x * x;
}
```

Generating a `future` object that will handle the task of squaring a given integer is accomplished by calling the `async` function with the function name and its argument. In practice, the `async` function can pass as many arguments as required to the function it is asked to run concurrently. To calculate the square of 5 as a task would therefore be written as:

```
#include <future>

// . . .

int i = 5;
```

```
auto ftr = std::async(square_val, i);
cout << std::format("Square of {} is {}\n", i, ftr.get());
```

Illustration: include *function* and *argument* labels above, as were in the MSFT Word document (pasted—can't do that in Asciidoc).

The resulting `ftr` object represents the result of the computation performed concurrently by the function passed to `async(.)`; that result will become available once that task has been completed. At this point, the result can be obtained by calling the `get()` member function defined on the `future` object; calling `get()` blocks the calling thread until the task's execution is completed.

This alone is of course not all that interesting, and this example will be much slower than the sequential equivalent due to the simplicity of the computation performed asynchronously, but let us leave this detail aside for the moment. We will soon see the calling code could essentially remain as simple for more complex asynchronous computations where improvied performance is measurable.

Continuing with the integer square theme, we could implement a varation that calculates the squares of a container of integers by calling the `async(.)` function for each number and executing each task asynchronously. A convenient way to do this is by placing them in a vector of `future` objects as each is created, and then iterating through and calling the `get()` member function to obtain each result. For example:

```
std::vector<int> x(25);
std::iota(x.begin(), x.end(), 0);     // 0, 1, 2, ..., 24
std::vector<std::future<int>> v;

for (auto k : x)
{
   v.emplace_back(std::async(square_val, k));
}

std::vector<int> y(v.size());

std::ranges::transform(v, y.begin(), [](std::future<int> &fut){
   return fut.get();
});
```

The vector `y` now contains the square of each of the integers in the vector `x`. This particular example again will not yield any performance gains, but it sets up a form we can use next in the case of Monte Carlo option pricing, where the results will show significant speed improvements over the sequential implementation presented earlier in the original discussion of Monte Carlo methods.

Monte Carlo Option Pricing Revisited

We can now return to our earlier Monte Carlo option pricing example and set it up to run in parallel. While in practice we would want to avoid duplicated code, for the purposes of demonstration, a parallelized version of the `calc_price(.)` member function is reimplemented in a new `calc_price_par(.)` function. This new function will calculate the price of the option, given the same inputs as the non-parallelized case.

As the random price scenarios "don't care" about any of the others, they are embarrassingly parallelizable and are thus good candidates for using `async(.)` and `future` to generate each scenario as a parallel task. As such, we do not need to be concerned with data races.

The interesting logic will again be executed only if the time left to expiration is positive, and if the barrier has not already been trivially crossed at the outset. As before, we will first generate a `vector` of seeds from a uniform integer distribution, and define a `vector` to contain the discounted payoffs.

Now just concentrating on the nontrivial case where $t > 0$, and where a barrier (if it exists) has not been crossed (yet), the code starts out as before with the random uniform integer generator created to provide a unique seed value for each scenario, a `vector` to hold the discounted payoff values, and the calculation of the constant discount factor to be applied to each terminal payoff.

```
double MCOptionValuation::calc_price_par(double spot, unsigned num_scenarios, unsigned unif_start_seed)
{
    // . . .

    if (opt_.time_to_expiration() > 0)
    {
        using std::vector;

        std::mt19937_64 mt_unif{ unif_start_seed };
        std::uniform_int_distribution<unsigned> unif_int_dist{};

        vector<double> discounted_payoffs;
        discounted_payoffs.reserve(num_scenarios);
        const double disc_factor = std::exp(-int_rate_ * opt_.time_to_expiration());

        vector<std::future<vector<double>>> ftrs;                    // (1)
        ftrs.reserve(num_scenarios);

        if (barrier_hit != true)
        {
            for (unsigned i = 0; i < num_scenarios; ++i)
            {
                EquityPriceGenerator epg{ spot, time_steps_, opt_.time_to_expiration(), vol_,
                    int_rate_, div_rate_ };

                // Each scenario is now asynchronous:
```

```
                    ftrs.emplace_back(epg, unif_int_dist(mt_unif)));     // (2)
                }

                for (auto& ftr : ftrs)
                {
                    vector scenario = ftr.get();                         // (3)

                    switch (barrier_type_)
                    {
                        // . . .
                    }

                    // . . .

                }
            }

            return (1.0 / num_scenarios) * std::accumulate(discounted_payoffs.cbegin(),
                discounted_payoffs.cend(), 0.0);
        }
        else
        {
            return opt_.option_payoff(spot);
        }
    }
```

The only differences are minor changes as follows. Instead of generating each scenario serially for each seed, we can spawn these in independent tasks that run asynchronously, allowing the entire set of scenarios to be generated more quickly. This is done by first creating a `vector` to hold a set of `std::future` objects handling each scenario (1):

```
vector<std::future<vector<double>>> ftrs;                                // (1)
```

Then, in (2), the scenarios are generated concurrently on individual threads by invoking the `std::async` function:

```
// Next seed = unif_int_dist(mt_unif)
ftrs.emplace_back(std::async(epg, unif_int_dist(mt_unif)));          // (2)
```

Now, in (3), each scenario is obtained by calling the `get()` member function on each `future` object in the `ftrs` vector. Once we have each `scenario`, the `switch`/`case` statement remains the same as it was in the serial `calc_price(.)` implementation we developed previously.

```
for (auto& ftr : ftrs)
{
      vector scenario = ftrs[i].get();          // (3)
      switch (barrier_type_)
      {
            // . . .
      }

      // . . .
}
```

By making these simple changes in just the three locations of the code described above, the performance can be improved remarkably; however, there are some caveats involved. These are summarized in the next section.

Caveats and Performance

Similar to parallel STL algorithms, one should not just blindly apply the use of `std::async` and expect automatic improvements in performance. Indeed, doing so may or may not result in an improvement, and switching from a single-threaded execution model to a multithreaded equivalent could even lead to performance losses. On the other hand, past a certain point, performance improvements can be quite substantial.

The following tables show the results for a down-and-out put option, with the following data:

Table 5-2. Down and Out Barrier Put Option Data

Spot	Strike	Risk-Free Rate	Volatility	Dividend Rate	Barrier Level
100	105	5%	25%	8%	70.50

Two cases of expirations were used, the first being one year, and the second with 10 years (more typically found in long-term guaranteed investment products and interest rate derivatives).

These tests were again performed on the same eight-core processor hardware as in the parallel STL algorithm examples. The results are as follows

(Add plot here in R as well to visualize the results):

Table 5-3. Down and Out Barrier Put Option Performance Comparison (One Year Expiration)

Num Time Steps	Num Scenarios	No async (ms)	With async (ms)	Speedup	Option Value
12	20000	36.7760	63.1752	-41.79%	7.54
12	50000	85.3634	185.5319	-53.99%	7.46
120	20000	127.257	88.2092	44.27%	6.75
120	50000	352.2114	230.6709	52.69%	6.75
360	20000	339.5314	61.7573	449.78%	6.58
360	50000	796.6583	179.3042	344.31%	6.56

Table 5-4. Down and Out Barrier Put Option Performance Comparison (10-Year Expiration)

Num Time Steps	Num Scenarios	No async (ms)	With async (ms)	Speedup	Option Value
10	20000	32.4281	45.2584	-28.35%	0.49
10	50000	84.5093	133.7967	-36.84%	0.48
40	20000	62.2338	49.5326	25.64%	0.49
40	50000	135.6066	116.9286	15.97%	0.48
120	20000	112.6142	48.4432	132.47%	0.36
120	50000	295.2015	162.1747	82.03%	0.37
520	20000	431.6498	146.6053	194.43%	0.30
520	50000	1096.2874	314.919	248.12%	0.30
3600	20000	2325.4275	449.539	417.29%	0.27
3600	50000	5727.0185	824.4835	594.62%	0.29

As can be seen in the results, the threaded cases for smaller numbers of time steps actually result in a slowdown in performance. However, as these are increased, we can start to see some rather significant speed improvements. Considering also that the closed-form theoretical price for the one-year option is \$6.29, and for the 10-year, \$0.26 (these can again be verified using the tools **{12}** and **{13}**), greater numbers of time steps become necessary for reasonable convergence.

One more consideration is that sensitivities often need to be calculated when valuing options. In many cases, this involves recalculating the option values with Monte Carlo methods using slightly perturbed values of the underlying price (delta), interest rate (rho), time to expiration (theta), etc. To estimate the delta value, one can approximate the first derivative value as follows, where S represents the underlying share price, δS (lower case delta) represents a small shift in the price (eg 0.5%), and V represents the value of the option for a given underlying price and all other parameters held equal **{18}**:

$$\Delta = \frac{V(S+\delta S)-V(S-\delta S)}{2\delta S}$$

As such, this means we have to run Monte Carlo simulations to calculate the option values two additional times, $V(S + \delta S)$ and $V(S - \delta S)$, so performance improvements of even a "modest", say 200%, increase in speed can make a big difference, given similar calculations would need to be run for just the remaining first-order sensitivities alone, rho, vega, theta, plus the second-order gamma calculations. These calculations can become unstable, however, in cases of discontinuous payoffs such as with barriers. In these cases, more sophisticated methods are typically warranted. **{19}** provides a detailed discussion.

The results are quite remarkable considering that, similar to our earlier examples with parallel algorithms, we were able to realize these performance gains with very minimal modification, and without having to manually program code associated with thread management. In fact, these performance gains were obtained with very simple changes to just two lines of the the code.

One observation that you might have is that there is no such thing as an exchange-traded equity option with an expiration time of ten years. This is true; however, as alluded to at the outset, there are examples where longer term contracts become relevant. One case is with complex structured interest rate derivatives that will have a series of fixed exercise dates over long periods of time, some of which also involve swapping currencies or that are embedded in credit derivatives.

Another example is liability calculations for guaranteed investment products typically offered by insurance companies, such as variable annuities. These contracts are essentially synthetic long-term put options with various complex riders such as ratchets, guaranteed return roll-ups, withdrawal benefits, and death benefits. They can span a period even on the order of 30 years and can even require nested simulations, pumping up the number of simulations and computational complexity dramatically. Monte Carlo is the method of choice for these types of models, and having task-based concurrency tools now available with C++ means much shorter implementation times, and portable implementations.

Finally, there are a couple of ways to consider to improve performance even further. First, as may seem obvious, would be by increasing the number of processors. This can be seen, for example, in a presentation by the author in 2019, where similar Monte Carlo option pricing models were run on a 20-core virtual machine **{20}**.

There is also a variety of variance reduction methods such as antithetic sampling and control variates that could be implemented to yield increased efficiency. Although we will not pursue this topic here, the previous discussion should provide you with a basic framework which you can fairly easily extend to include variance reduction code. Comprehensive coverage of variance reduction methods is also available in the previously cited book *Monte Carlo Methods in Financial Engineering*, by Paul Glasserman **{21}**.

Concluding Remarks on `async` and `future`

For completeness, it should be noted that `future` and `async(.)` have received a substantial amount of criticism because of certain issues relating to more complex situations. However, for models based on

embarrassingly parallelizable processes, and thus where data races are not an issue, `future` and `async(.)` provide a simple and reliable means for implementation of multithreaded tasks. This means potentially reaping substantial performance improvements without having to deal with the extra work and complications that can come with manual implementation of thread management.

A recommendation for making `async(.)` more reliable **{23}** is to supply a *launch policy* with explicit instructions to run each task on a different thread. This may seem counterintuitive, but the default can potentially run in a deferred mode, as there are still single core machines out there, and `async(.)` can run into trouble with these otherwise.

In our case, on a multicore platform, this would mean when generating each new scenario in our Monte Carlo option pricing model, instead of

```
ftrs.emplace_back(std::async(epg, unif_int_dist(mt_unif)));
```

we would instead include the launch policy as an additional argument, `std::launch::async`, as shown here:

```
ftrs.emplace_back(std::async(std::launch::async, epg, unif_int_dist(mt_unif)));
```

The details behind this get complicated and are beyond the scope of this book, but more information can be found in Scott Meyers' *Effective Modern C++* **{24}**.

Future enhancements are planned for a more robust concurrency library in the Standard, but it appears `async(.)` will remain as a no-frills option, highly suitable for embarrassingly parallelizable processes such as the Monte Carlo model presented here. It is recommended, however, to use the launch policy as noted above.

Summary

With perhaps the exception of parallel STL algorithms, the title of this chapter could have been titled "useful modern features of C++ you probably have never heard of". Having distributional random number generation as part of the Standard Library is a huge improvement for quantitative financial developers. For Monte Carlo simulation alone, as shown earlier, this means the very common tasks of implementing a reliable standard normal random number generator is already provided and well-tested.

`std::future` and `std::async` provide a ready-made method for implementing embarrassingly parallel processes that are ubiquitous in finance, without requiring the programmer to manually write lines of complex code responsible for creating thread pools, implementing parallel execution, and then deactivating the threads. It also ensures the code remains platform-independent, without reliance on third party commercial libraries. Goosing performance further is an option, using newer threading features that have

been added to the Standard in more recent years, but this would inevitably introduce the cost of longer development times and higher maintenance requirements. It would come down to weighing the benefits—likely diminishing but perhaps still desired—vs the additional development time, testing, and maintenance.

Parallel STL algorithms, however, have received a lot of attention, and they also can provide an enormous peformance benefit considering how easy they are to use, simply adding an execution policy argument to a previously existing algorithm, or utilizing new parallel numeric algorithms such as `std::reduce` and `std::transform_reduce`. Furthermore, parallel "overloads of all the existing algorithms in `std::ranges` that have a parallel overload in" the Standard Library are also in discussion for a future release **{25}**.

References

{1} Makoto Matsumoto and Takuji Nishimura, *Mersenne Twister: A 623-dimensionally Equidistributed Uniform Random Number Generator*, ACM Transactions on Modeling and Computer Simulation, Volume 8, Issue 1, 1998 (*http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/ARTICLES/mt.pdf*)

{2} Paul Glasserman, *Monte Carlo Methods in Financial Engineering*, Section 2.3.2, Springer-Verlag, 2004

{3} Walter E Brown, *Random Number Generation in C++11*, *https://wg21.link/n3551*

{4} Author example using the Clang option on Compiler Explorer(*https://godbolt.org/z/8r7ET7va7*)

{5} op cit Glasserman, Section 3.1

{6} Peter Jäckel, *Monte Carlo Methods in Finance*, Section 7.5, Wiley Finance, 2002

{7} cppreference, *Pseudo-random number generation*, *https://en.cppreference.com/w/cpp/numeric/random*

{8} ibid

{9} Jaekle and Tomasini, Trading Systems (2E) (Ch 4)

{10'} Kevin Dowd, *Beyond Value at Risk: The New Science of Risk Management*, Ch 2, Sec 2.2: Parametric VaR, pp 42-43

{10} Peter James, *Option Theory*, Wiley Finance, 2003, Chapter 10 Section 10.2

{11} James, op cit, Chapter 15, Section 15.2

{12} Robert McDonald, *Option Pricing Functions to Accompany Derivatives Markets* (vignette for the *derivmkts* R package), 2022-04-11, *https://rdrr.io/cran/derivmkts/f/inst/doc/derivmkts-vignette.pdf*

{13} Barrier Option Pricing, Coggit Free Tools, *https://coggit.com/freetools*

{14} Lucian Radu Teodorescu, *A Case Against Blind Use of C++ Parallel Algorithms*, *https://accu.org/journals/overload/29/161/teodorescu/*, in particular see "Problem 4: small datasets are not good for parallelisation"

{15} (Hanson, CppCon 2019)

{16} *C++ Parallel Algorithms*, NVIDIA Accelerated Computing: HPC SDK Documentation, *https://docs.nvidia.com/hpc-sdk/compilers/c++-parallel-algorithms/index.html*

{17} Hans-J. Boehm, *Threads Cannot be Implemented as a Library*, HP Tech Reports, November 12, 2004, *https://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf*

{18} James, op cit, Sec 10.4, p 135

{19} Glasserman, op cit, Ch 7, Sec 7.2

{20} Hanson, op cit

{21} Glasserman, op cit, Ch 4

{23} Eli Bendersky, *The Promises and Challenges of std::async Task-based Parallelism in C++11*,

{24} Scott Meyers, *Effective Modern C++*, O'Reilly (2015) (Put O'Reilly link here)

{25} Revzin, Hoekstra, and Song, *A Plan for C++23 Ranges (Tier 2)*, *https://wg21.link/p2214#tier-2*, April 2016, *https://dzone.com/articles/the-promises-and-challenges-of-stdasync-task-based*

Extensive coverage of longer-term guaranteed investment products typically offered through life insurance companies, for which Monte Carlo methods are frequently employed in hedging and risk management, can be found in the book *Investment Guarantees*, by Mary Hardy (Wiley Finance, 2003).

For a formal definition of data races in C++, see *https://wg21.link/intro.races*

Plots of drawdowns were done using the base R `plot(.)` function.

Plots of equity price scenarios were done using the xts R package (cite authors).