CHAPTER 7 Parallel Implementation

In this chapter, we extend the code from the previous chapter with parallel simulations, applying everything we learned in Part I. It is generally not an easy task to parallelize an existing engine, because odds are that various parts were developed without thread safety in mind. We developed our engine from scratch in the previous chapter, knowing that we would eventually extend it with parallelism, so our work here is substantially less challenging than in real life.

It follows that we will not modify base or concrete models or products in any way, but we will need a substantial extension in the base and concrete RNGs.

The parallel algorithm is implemented in a parallel version of the template algorithm of the previous chapter, $mcParallelSimul()$, also in mcBase.h in our repository.

7.1 PARALLEL CODE AND SKIP AHEAD

Parallel simulation design

With serial path-wise MC, we processed the paths sequentially in a loop, the processing of every path consisting in the three following steps:

1. Generate a random Gaussian vector in dimension D with a call to $RNG :: nextG()$.
2. Consume the random vector to generate a scenario for the market observations on the event dates with a call to $Model :: generatePath()$.
3. Compute the payoffs over that path with a call to $Product :: payoffs()$.

We now process different paths in parallel. We know how to do this with the thread pool of Section 3.18 and the instructions on page 108. We spawn the paths for parallel processing into the thread pool, with the syntax:[1]

The thread pool schedules the parallel execution of the task and immediately returns a future that the main thread uses to monitor the status of the task. After the main thread finished dispatching the tasks, it waits for the tasks to complete and helps the worker threads executing the tasks in the meantime, with the syntax:

where is the vector of the futures associated to the tasks.

It follows that the parallel code is very similar to from the previous chapter, with the exception that the three lines which constitute the processing of a path are executed in parallel in the thread pool instead of sequentially on the main thread.

In reality, each task will consist not of a single path, but a batch of paths. We refer to the discussion on page 112, where we concluded that it is best to:

- spawn a large number of parallel tasks compared to the number of available cores, to take advantage of load balancing; and
- make tasks large enough so they take at least 0.1 ms to 1 ms of CPU time, so that the admin cost of sending the task to the pool remains negligible compared to its execution cost.

We estimated that a path takes around 6 microseconds of CPU time, and therefore default to batches of 64 paths each.

Thread safe design

If we are going to process different paths from different threads, the processing must be thread safe. , and must be safe to call concurrently without interference. We knew that when we designed the simulation library, so we carefully coded the methods and (correctly) const. This is an example of thread safe design, since we did not, and will not, apply any lock in our simulation code. Those two methods can be safely called on the same model and product from different threads.

, however, is not a const method: it modifies the state of the generator when it produces numbers. We cannot use the same RNG from different threads concurrently. The RNG is a *mutable object*, as defined on page 109, together with detailed instructions for making copies of such objects for every thread, and work with the executing thread's own copy from concurrent code. Our code scrupulously follows these instructions.

In addition to the RNG, we have two mutable objects on the workspace: a preallocated vector to hold the Gaussian numbers, and a preallocated vector of samples, or scenario, filled by

and consumed by

. We cope with these mutable objects in the exact same manner.

Skip ahead

All that precedes is a direct application of the parallel computing idioms from Part I to the embarrassingly parallel code in

from the previous chapter. Readers are encouraged to review Section 3.19.

The only real difficulty is the following: we need the concurrent code to process in parallel the *same* paths that the serial code processes sequentially. With random sampling, this is nice to have. The multi-threaded algorithm produces the exact same results as the single-threaded version for a finite number of paths, and not only on convergence. With Sobol, this is *compulsory*. The low discrepancy of a Sobol sequence is only guaranteed when the points are sampled in a sequence.

The problem is that RNGs draw random numbers or vectors in a sequence, updating internal state with every new draw. The state that produces the random vector number is the result of the prior draws. To produce the same results in parallel, RNGs must acquire the ability to *skip ahead* to the state efficiently.

One obvious way to implement skip ahead is draw samples and throw them away. This is obviously an inefficient process, linear in .
Many concrete RNGs can be skipped ahead much more efficiently, but this is entirely dependent on the RNG. In the case of both mrg32k3a and Sobol, a skip ahead of *logarithmic* complexity can be implemented.

For now, we add a virtual skip ahead method to our base RNG class, with a default implementation that generates and discards the first draws. Concrete RNGs may override it with a more efficient skip ahead, when available:

We can proceed with the development and testing of the parallel template algorithm, and override with fast skip ahead routines in the concrete RNGs thereafter.

The parallel template algorithm

The parallel template algorithm consists in rather minor modifications to the serial version, following to the letter the instructions of Section 3.19.

We can execute the code straightaway, having already coded the interface in the entry point function in Section 6.6. The code works nicely, to execute it occupies 100% of the hardware threads, and it produces the *exact same* results as its single-threaded counterpart, approximately 10 to 20 times *slower*. The default skip ahead is so expensive that it exceeds by far the benefits of parallelism. We must now implement fast skip ahead algorithms for our concrete RNGs.

7.2 SKIP AHEAD WITH MRG32K3A

The transition equations on page 201 for mrg32k3a may be written in the compact matrix form:

where:

and:

Since moduli and matrix products are commutative, it immediately follows that the state obtained after the production of numbers:[2]

is:

It follows that mrg32k3a is skipped ahead to state by the left multiplication of the initial state vectors and by the matrices and (and the application of the moduli).

A classical and well-known logarithmic algorithm exists to compute the power matrix . Consider the binary decomposition of , that is the unique set of such that

Then:

where we denoted . The s trivially satisfy:

Hence, all the s (and hence,) are computed in a number of matrix products corresponding to the number of bits in the binary decomposition of , that is , a maximum of 32 for a 32-bit integer. The skip ahead completes in logarithmic time (17 matrix products in dimension skipping 100,000 numbers).

A practical implementation must overcome the additional difficulty of overflow consecutive to the repeated multiplication of big numbers. We note that the product of two 32-bit integers cannot exceed 64 bits. Therefore, we store results in 64-bit unsigned long longs, and nest moduli after every scalar operation. This is slower than the double-based implementation of the generator in the previous chapter, but it doesn't matter for a logarithmic algorithm. First, we develop custom matrix products as private static methods of mrg32k3a, and a private method to skip numbers with the algorithm we just described:

Next, we implement the public override , taking into account the vector and antithetic logics:

7.3 SKIP AHEAD WITH SOBOL

Recall from the equation on page 207 that Sobol's state is a (xor) combination of direction numbers with coefficients 0 or 1:

and it immediately follows that Sobol is skipped to state in logarithmic time.

In addition, we have seen that flicks in and out every two numbers, starting with number 1. It follows that when:

is odd, zero otherwise. flicks every four numbers, starting with number 2, so when:

is odd, zero otherwise. flicks every eight numbers, starting with number 4, so when:

is odd, zero otherwise. flicks every numbers, starting with number , so when:

is odd, zero otherwise. And it follows that:

where the sum means xor-sum and:

The code is listed below:

7.4 RESULTS

The parallel code in this chapter brings together everything we learned so far. It produces the exact same results as the sequential code, faster by the number of available CPU cores. On our iMac Pro (8 cores), we exactly reproduce the examples of Section 6.6 in 350 ms with either mrg32k3a or Sobol, with a perfect parallel efficiency.[3] It is remarkable that we can run 500,000 simulations over 156 time steps in something like a quarter of a second.

Timing is very satisfactory, as is parallel efficiency. Still, with 350 ms pricing, it would take several minutes to compute a risk report of all the sensitivities to the 1,800 (30 × 60) local volatilities with traditional finite differences. In the final and most important part of this publication, we will learn techniques to make this happen in *half a second* without loss of accuracy.

NOTES

1 We must start the thread pool with a call to:

when the application launches, and stop it with a call to:

when the application closes. In an Excel-based application, we start the pool in , right after the registration of our exported functions. In a console application, we would do that in the first lines of . The thread pool always remains available in the background, and it was designed so that it doesn't consume any resources when unused.

---

2 Careful that mrg32k3a works with numbers, not vectors.

3 Acceleration is superior to the number eight of physical cores thanks to hyper-threading.