Introduction

This part is a self-contained tutorial and reference on high-performance programming in C++, with a special focus on parallel and concurrent programming. The second part applies the notions and constructs developed here to build a generic parallel financial simulation library.

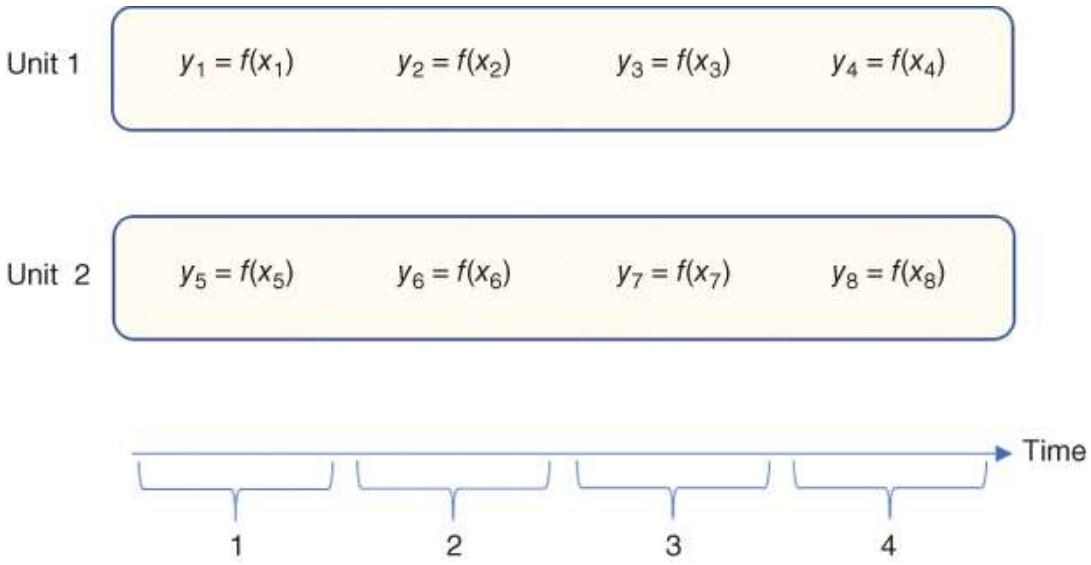
A working knowledge of C++ is expected from readers. We cover modern and parallel C++11, and we illustrate the application of many STL (standard template library) data structures and algorithms, but we don't review the basic syntax, constructs, and idioms of C++. Readers unfamiliar with basic C++ should read an introductory textbook before proceeding. Mark Joshi's website http://www.markjoshi.com/RecommendedBooks.html#C contains a list of recommended C++ textbooks.

Readers already familiar with advanced C++ and concurrent programming, or those using different material, such as Anthony Williams' [15], may skip this part and move on to Parts II and III. They will need the *thread pool*, which we build in Section 3.18 and use in the rest of the publication. The code for the thread pool is in ThreadPool.h, ThreadPool.cpp, and ConcurrentQueue.h in our repository.

PARALLEL ALGORITHMS

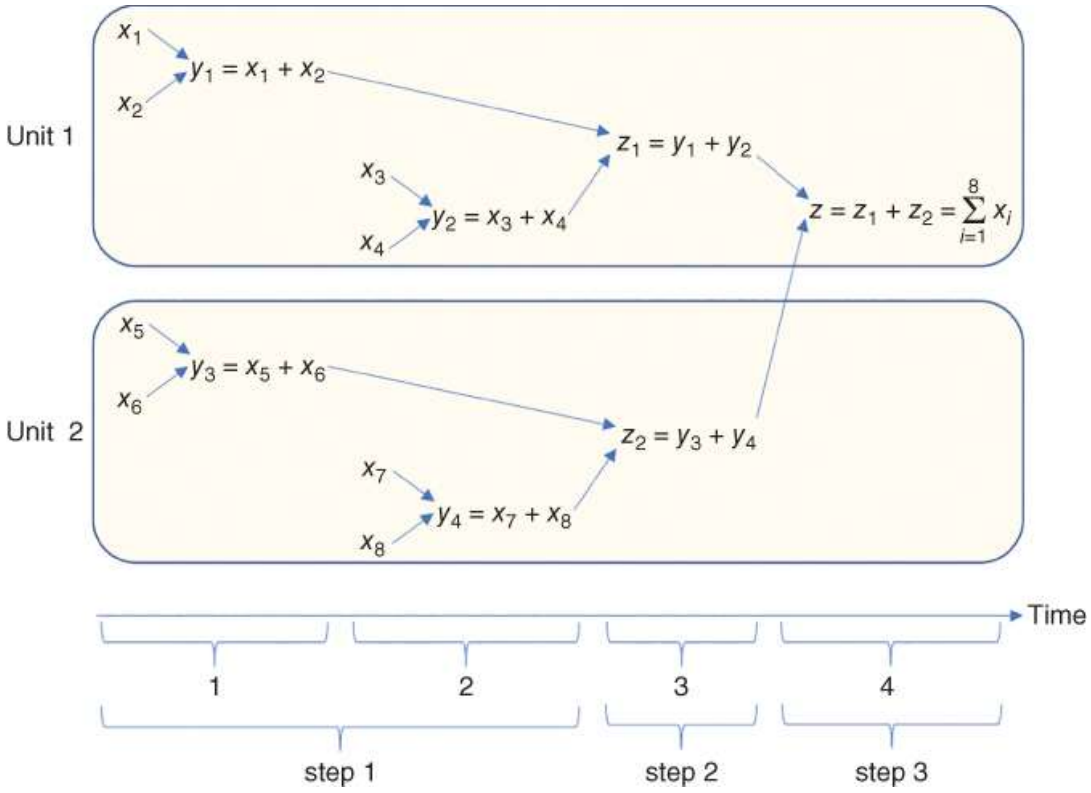
Parallel programming may allow calculations to complete faster by computing various parts simultaneously on multiple processing units. The improvement in speed is at best linear in the number of units. To

hit a linear improvement (or any improvement at all), parallel algorithms must be carefully designed and implemented. One simple, yet fundamental example is a *transformation*, implemented in the $transform()$ function of the C++ standard library, where some function f is applied to all the elements $\{x_i\}$ in a source collection S to produce a destination collection $D = \{y_i = f(x_i)\}$. Such transformation is easily transposed in parallel with the "divide-and-conquer" idiom: partition S into a number of subsets $\{S_j\}$ and process the transformation of the different subsets, simultaneously, on different units. The chart below illustrates the idiom for the transformation of 8 elements on 2 units.



The 8 transformations are processed in the time of 4. Parallel processing cuts time by the number of units, a *linear* speed-up. With n transformations on p units, each unit evaluates f n/p times, and the units work simultaneously, so that the entire transformation completes in the time of n/p evaluations, as opposed to n evaluations in the serial version. In addition, this linear improvement is easily achieved in practice, assuming f computes an output y out of an input x without any side effect (like logging into a console window or a file, or updating a static cache).[1] Such problems that are trivially transposed and implemented in parallel are called "embarrassingly parallel."

Another classical family of fundamental algorithms is the *reduction*, implemented in the standard C++ $accumulate()$ function, which computes an aggregate value (sum, product, average, variance...) from a collection of elements. One simple example is the sum $z = \sum_{i=1}^{n} x_i$. In order to compute z in parallel, we can compute partial sums over subsets, which leaves us with a lower number of partial sums to sum-up. A parallel reduction is therefore recursive in nature: we compute partial sums, then partial sums of partial sums, and so forth, until we get a final sum. The chart below illustrates the sum of 8 terms on 2 units:



A reduction of 8 elements is, again, completed in the time of 4. A parallel reduction also produces a linear improvement. But the algorithm is no longer trivial; it involves 3 steps in a strict sequential order, and logic additional to the serial version. In general, to reduce n elements in parallel on p units, assuming n is even, we partition $S = \{x_i, 1 \leq i \leq n\}$ into $n/2$ pairs and sum the 2 elements of the $n/2$ pairs in parallel. To process $n/2$ adds in parallel on p units takes the time of $n/2p$ serial adds. Denoting δ the time of a serial add, we

reduced S into $n/2$ partial sums in $(n/2p)\delta$ time. Assuming again that $n/2$ is even, we repeat and reduce the $n/2$ partial sums into $n/4$ pairwise sums in a time $(n/4p)\delta$. By recursion, and assuming that n is a power of 2: $n = 2^m$, the entire reduction completes in $m = log_2 n$ steps, where the step number i consists in $n/2^i$ pairwise adds on p units and takes $(n/2^i p)\delta$ time. The total time is therefore:

$$\sum_{i=1}^{m} \frac{n}{2^i p}\delta = \frac{n-1}{p}\delta$$

as opposed to $(n-1)\delta$ for the serial reduction, hence an improvement by p.

In a more general case where n is not necessarily a power of 2, we still achieve a linear improvement. To demonstrate this, we note that n can be uniquely decomposed into a sum of powers of 2:

$$n = \sum_{i=1}^{M} n_i, n_i = 2^{m_i}$$

This is called the *binary decomposition* of n.[2] We conduct parallel reductions over the M subsets of $n_i = 2^{m_i}$ elements, each one in time $\delta(n_i - 1)/p$, as explained above, a total time of $\delta(n - M)/p$. We repeat and reduce the M partial sums in subsets of sizes the binary decomposition of M, in a time $\delta(M - M_2)/p$, where M_2 is the number of terms in the binary decomposition of M, reduce the M_2 results, and repeat recursively until we obtain one final reduction of the whole set, in a total time $\delta(n-1)/p$.

As for the transformation, we get a linear improvement, but with a less trivial algorithm, both to comprehend and design, and to implement in practice. All units must complete a step before the next step begins, so units must wait on one another, and the algorithm must incorporate some form of *synchronization*. Synchronization overhead, combined with the fact that the last steps involve fewer elements to sum-up than available units, makes a linear speed-up hard to achieve in practice.

Of course, this classical algorithm applies to all forms of reduction, not only sums, but also products, dot products, variance and covariance estimates, etc. Because it applies to dot products, it also applies to matrix products, where every cell in the result matrix is a dot product.

We have introduced two fundamental examples. One is embarrassingly parallel and fully benefits from parallelism without difficulty. The other one takes work and rarely achieves full parallel efficiency. In finance, derivatives risk management mainly involves two families of valuation algorithms: Monte-Carlo (MC) simulations and finite difference methods (FDM). MC simulations compute the average outcome over many simulated evolutions of the world. FDM computes the same result as the solution of a partial differential equation, numerically, over a grid. MC simulations are slow, heavy, and embarrassingly parallel. They are also applicable in a vast number of contexts. MC is therefore, by far, the most widely used method in modern finance; it is easy to implement in parallel, and the linear speed improvement makes a major difference due to the slow performance of the serial version. Part II is dedicated to the implementation of parallel MC. In contrast, FDM is light and fast, but applicable in a limited number of situations. FDM is less trivial to implement in parallel, and the resulting acceleration may not make a significant difference, FDM performing remarkably fast in its serial form. FDM and its parallel implementation are not covered in this text.

THE MANY FLAVORS OF PARALLEL PROGRAMMING

Distributed and concurrent programming

Parallel programming traditionally comes in two flavors: distributed and concurrent. The distributed flavor divides work between multiple *processes*. Processes run in parallel in separate memory spaces, the operating system (OS) *scheduling* their execution over the available processing units. Distributed computing is safe by design because pro-

cesses cannot interfere with one another. It is also scalable to a large number of processing units, because different processes may run on the same computer or on multiple computers on a network. Since processes cannot communicate through memory, their synchronization is achieved through *messaging*. Distributed computing is very flexible. The cooperation of different nodes through messaging can accommodate many parallel designs.

Distributed programming is not implemented in standard C++ and requires specialized hardware and software.[3] The creation and the management of processes takes substantial overhead. Processes cannot share memory, so the entire context must be copied on every node. Distributed programming is best suited for *high-level* parallelism, like the distribution of a derivatives book, by transaction, over multiple machines in a data center. A lighter form of parallel computing, called *shared memory parallelism* or *concurrent programming*, is best suited for the implementation of parallelism in the lower level valuation and risk algorithms themselves.

Concurrent programming divides work between multiple *threads* that run on the same process and share common memory space. Threads are light forms of processes. Their execution is also scheduled by the OS over the available processing units, but limited to a single computer. The overhead for their creation and management is orders of magnitude lower. They share context and communicate directly through memory. The management of threads in C++ is standard and portable since C++11. All modern compilers, including Visual Studio since 2010, incorporate the Standard Threading Library out of the box. The threading library provides all the necessary tools and constructs for the manipulation of threads. It is also well documented, in particular in the book [15] written by one of its developers.

Concurrent programming is generally the preferred framework for the implementation of parallel algorithms. This text focuses on concurrent programming and does not cover distributed programming.

Concurrent computing is not without constraints: it is limited to the processing units available on one computer, and, contrary to the distributed memory model, it is not safe by design. Threads may interfere with one another in shared memory, causing severe problems like crashes, incorrect results, slowdowns, or deadlocks, and it is the developer's responsibility to implement thread safe design and correct synchronization to prevent these. The Standard Threading Library provides a framework and multiple tools to facilitate correct concurrent programming. This library and its application to the development of safe, efficient parallel algorithms, is the main subject of this part.

CPU and GPU programming

A particular form of concurrent computing is with GPU (Graphic Processing Unit) parallelism. Modern GPUs provide a massive number of processing units for limited hardware costs. GPUs were initially designed to process the computation of two- and three-dimensional graphics, and later evolved to offer general-purpose computing capabilities. Nvidia, in particular, offers a freely available C++-like language, CUDA (Compute Unified Device Architecture), for programming their chips. GPU programming is implemented today in many scientific and commercial applications besides graphic design and video games. In particular, GPU accelerates many machine learning and financial risk management softwares. Reliable sources in the financial industry report speed improvements of order 50× for production Monte Carlo code; see for instance [16].

GPU programming is evidently not standard C++. In addition to specialized hardware and software, effective GPU programming requires algorithms to be written in a specific manner, down to low level, to accommodate the design and constraints of GPUs. GPU parallelism may come with unbeatable hardware cost; it is also subject to prohibitive development cost. For this reason, GPUs somewhat fell out of favor recently in the financial industry; see for instance [17].

CPU (Central Processing Unit) manufacturers like Intel have been systematically increasing the number of cores on their chips in recent years, so that CPUs may compute with GPUs, without the need for specialized GPU programming. High-end modern workstations now include up to 48 cores (up to 18 on Apple's slim iMac Pro). CPU cores are truly separate units, so different cores may conduct independent, unrelated work concurrently. GPU cores are not quite that flexible. CPU cores are also typically substantially faster than their GPU counterparts. More importantly, CPU parallelism is programmed in standard C++.

In addition, it is very challenging to effectively run memory-intensive algorithms, like AAD, on GPUs. Algorithmic adjoint differentiation (AAD) produces risk sensitivities in constant time (see our dedicated Part III), something no amount of parallelism can achieve. AAD is the most effective acceleration implemented in the financial industry in the past decade.[4] It is not particularly difficult to implement AAD on parallel CPU. We do that in Part III. On the other hand, to our knowledge, AAD has not yet been convincingly implemented on GPU, despite recent encouraging results by Uwe Naumann from RWTH Aachen University.

For those reasons, we don't explore GPU parallelism further and focus on CPU concurrency. Readers interested in learning GPU programming are referred to one of the many available CUDA textbooks.

Multi-threading and SIMD programming

Finally, concurrent CPU programming itself splits into two distinct, mutually compatible forms of parallelism: multi-threading and SIMD.

Multi-threading (MT) consists in the concurrent execution of different parts of some calculation on different threads running on different cores. The cores are effectively independent CPUs, so they may run unrelated calculations concurrently without loss of performance.[5] For

example, the transformation of a collection can be multi-threaded by transforming different subsets over different threads. In this case, the different cores will execute the same sequence of instructions, but over different data. In a simple video game, the management of the player's star ship and the management of the enemies can be processed on two different threads. In that case, the two cores that execute the two threads conduct different, independent work. The game still runs twice as fast, providing a more satisfactory experience.

Multi-threading is a particularly flexible, efficient form of concurrent CPU programming, and the subject of Chapter 3. In Part II, it is applied to accelerate financial simulations.

By contrast, SIMD (Same Instruction Multiple Data) refers to the ability of every core in a modern CPU (or GPU) to apply the same instruction to multiple data, simultaneously. In a transformation, for instance, one single core could apply the function f to multiple xs simultaneously. SIMD applies CPU instructions to a vector of data at a time, as opposed to a single data. For this reason, it is also called “vectorization.” Modern mainstream CPUs (implementing AVX2) can process four doubles (32 bytes, 256 bits) at a time. Higher-end CPUs (AVX512) can process eight doubles (64 bytes, 512 bits) in a single instruction.

SIMD may be combined with multi-threading, in a process sometimes called “GPU on CPU” because GPUs implement a similar technology. For the transformation of a collection, multiple subsets can be processed on different threads running on different cores while every core applies SIMD to process multiple elements of the subset in a single instruction.

The theoretical acceleration from combining MT with SIMD is the product of the number of cores by the SIMD vector width. In a standard 8-core workstation with AVX2, this is 32. On an 18-core high-end iMac Pro (AVX512), this is a vertiginous 144, well above the acceleration reportedly obtained on GPU. But this is theoretical of course. MT

usually achieves linear acceleration as long as the code is correctly designed, as we will see in the rest of this book. SIMD acceleration is a different story.

SIMD is very restrictive: only simple functions applied to simple types may be vectorized. In addition, the data must be coalescent and properly aligned in memory. For example, a simple transformation $x \to \cos x$ of a set $S = \{x_i, 1 \leq i \leq n\}$ can be vectorized, and Visual Studio would indeed *automatically* vectorize a loop implementing this transformation. With a different function, like $x \to \log x \text{ if } x > 0 \text{ or } -\log -x \text{ otherwise}$, the transformation cannot (easily) be vectorized: the effective CPU instruction that applies to each data depends on whether the number is positive or negative; we are no longer in a *same* instruction multiple data context. Visual Studio would not vectorize the transformation.

SIMD is low level, close to the metal parallelism. It is not implemented in standard C++ or supported in standard libraries. SIMD is implemented in the compiler. Contrary to multi-threading, it is not scheduled at run time. It is at compile time, when the C++ code is turned into CPU instructions, that vectorization occurs or not. Standard compilers like Visual Studio offer little support for SIMD. Visual Studio automatically vectorizes simple loops, and may be set to reports which loops are vectorized and which are not. Other compilers, like Intel's, offer much more extensive support for vectorization in the form of settings and pragmas.[6]

Intel also offers its Integrated Performance Primitives (IPP) and Math Kernel Library (MKL) as a free download. These libraries include a vast number of vectorized mathematical functions and algorithms. With direct relevance for Monte-Carlo simulations, IPP offers a vectorized implementation of the inverse Normal distribution $ippsErfcInv\_64f\_A26()$. Hence, and even though Visual Studio supports SIMD for the occasional auto-vectorization of loops, an effec-

tive, systematic implementation of vectorization takes a specialized compiler and specialized third-party libraries.

In addition, in order to systematically benefit from SIMD, code must be written in a special way, so that mathematical operations are applied simultaneously to multiple data at low level. SIMD acceleration is *intrusive*. We will see in Part II that this is not the case for MT. We implement parallel simulations without modification of the model or the product code (as long as that code is *thread safe*). To vectorize simulations, we would need to rewrite the simulation code in models and products entirely.

A full SIMD acceleration is only achieved when the exact same CPU instructions are applied to different data. Whenever we have control flow in the code (if, while and friends), SIMD may be partially applied at best. Partial parallelism is a known pitfall in parallel programming. When perfect parallelism (acceleration by the number of units p) is achieved in a proportion μ of some algorithm, a proportion $1-\mu$ remains sequential, resulting in a global acceleration by a factor $p/[\mu+(1-\mu)p]$,[7] very different from the perhaps intuitive but completely wrong $p\mu$. The resulting acceleration is typically counter-intuitively weak. For instance, the perfect parallel implementation of 75% of an algorithm over 8 units results in an acceleration by a factor 2.9, terribly disappointing[8] and far from a perhaps mistakenly expected factor 6. Even when 90% of the algorithm is parallel with efficiency 100%, the resulting global acceleration is only 4.7, a parallel efficiency below 50%. Naive parallel implementation often causes bad surprises, and partial parallelism, in particular, generally produces disappointing results. Therefore, to achieve a linear acceleration with SIMD over complex, structured code is almost impossible. Besides, that theoretical limit is only 4× on most current CPUs. By contrast, a linear acceleration of up to 24× is easily achieved with MT. As long as the multi-threaded code is thread safe and lock-free, high-level multi-threading is, by construction, immune to partial parallelism. Of course, MT code is also vulnerable to sometimes challenging flaws and

inefficiencies, which we will encounter, and resolve, in the chapters of this book. We do achieve a linear acceleration for our simulation code, including differentiation with AAD.

Finally, and most importantly, it is extremely challenging to combine SIMD with AAD. Vectorized AAD may be the subject of a future paper, but it is way out of the scope of this book. In contrast, MT naturally combines with AAD, as demonstrated in detail in Chapter 12, where we apply AAD over multi-threaded simulations.

We see that the combination of MT with SIMD offers similar performance to GPU but suffers from the same constraints. This is not a coincidence. GPUs are essentially collections of a large number of slower cores with wide SIMD capabilities. Systematic vectorization requires rewriting calculation code, down to a low level, with a specialized compiler and specialized libraries. Of course, all of this also requires specialized skills. We will not cover SIMD in detail, and we will not attempt to write specialized code for SIMD acceleration. On the contrary, we will study MT in detail in Chapter 3, and implement it throughout the rest of this text.

We will also apply *casual* vectorization, whereby we write loops in a way to encourage the compiler to auto-vectorize them, for instance, by using the general algorithms of the standard C++ library, like $transform()$ or $accumulate()$, in place of hand-crafted loops, whenever possible. Those algorithms are optimized, including for vectorization when possible, and typically produce faster code. In addition, it makes our code more expressive and easier to read. It is therefore a general principle to apply STL (Standard Template Library) algorithms whenever possible. C++ programmers must know those algorithms and how to use them. We use a number of STL algorithms in the rest of the publication, and refer to [18] for a complete reference.

Readers interested in further learning SIMD are referred to Intel's documentation. For an overview and applications to finance, we refer to

[19]. To our knowledge, there exists no textbook on SIMD programming. MT programming is covered in detail in [15], and, in a more condensed form, in our Chapter 3.

Multi-threading programming frameworks

There exist many different frameworks for concurrent MT programming. The responsibility for the management of threads and their *scheduling* on hardware cores belongs to the operating system, and all major OS provide specific APIs for the creation and the management of threads. Those APIs, however, are generally not particularly user friendly, and obviously result in non-portable code. Vendors like Intel or Microsoft released multiple libraries that facilitate concurrent programming, but those products, while excellent, remain hardware or OS dependent.

Prior to C++11, two standardization initiatives were implemented to provide portable concurrent programming frameworks: OpenMP and Boost.Thread. OpenMP offers compile time concurrency and is supported by all modern compilers on all major platforms, including Visual Studio on Windows. Its consists of a number of compiler directives, called pragmas, that instruct the compiler to automatically generate parallel code for loops. OpenMP is particularly convenient for the parallelization of simple loops, and we provide an example in Chapter 1. OpenMP is available out of the box with most modern compilers, and therefore results in portable code. Using OpenMP is also amazingly simple, as demonstrated in our example.

For multi-threading complex, structured code, however, we need the full flexibility of a threading library. Boost.Thread provides such a library, with run-time parallelism, in the form of functions and objects that client code uses to create and manage threads and send tasks for concurrent execution.[9] Boost is available on all major OSs, and offers a consistent API for the management of threads and parallel tasks while encapsulating OS and hardware-specific logic. Many developers con-

sider Boost as “almost standard,” and applications written with Boost.Thread are often considered portable. However, compilers don't incorporate Boost out of the box. Applications using Boost.Thread must include Boost headers and a to Boost libraries. This is not a particularly simple, user friendly exercise, especially on Windows. It is also hard work to update Boost libraries in a vast project, for instance, when upgrading to a new version of Visual Studio.

Since C++11, C++ comes with a Standard Threading Library, which is essentially a port of Boost.Thread, better integrated within the C++ language, and consistent with other C++11 innovations and design patterns. Contrary to Boost.Thread, the Standard Threading Library is provided out of the box with all modern compilers, including Visual Studio, without the need to install or a against a third-party library. Applications using the Standard Threading Library are fully portable across compilers, OS, and hardware. In addition, as part of C++11 standard, this library is well documented, and the subject of numerous dedicated books, the best in our opinion being [15]. The Standard Threading Library is our preferred way of implementing parallelism. With very few exceptions, all the algorithms and developments that follow use this library.

WRITING MULTI-THREADED PROGRAMS

So, there exists many flavors of parallel computing and a wide choice of hardware and software environments for a practical implementation of each. Distributed, concurrent, and SIMD processing are not mutually exclusive. They form a hierarchy from the highest to the lowest level of parallelism. In Chapter 1, we implement a matrix product by multi-threading over the outermost loop and vectorizing the innermost loop. In a financial context, the risk of a bank's trading books can be split into multiple portfolios distributed over many computers. Each computer that calculates a portfolio may distribute multiple tasks concurrently across its cores. The calculation code that executes on every core may be written and compiled in a way to enable SIMD.

Similarly, banks using GPUs distribute their trading book across several machines, and further split the sub-books across the multiple cores on a computer, each core controlling one GPU that implements the risk algorithm. Maximum performance is obtained with the combination of multiple levels of parallelism. For the purpose of this publication, we made the choice, motivated by the many reasons enumerated above, to cover multi-threading with the Standard Threading Library.

As Moore's law has been showing signs of exhaustion over the past decade with CPU speeds limiting around 4GHz, chip manufacturers have been multiplying the number of cores in a chip as an alternative means of improving performance. However, whereas a program automatically runs faster on a faster chip, a single-threaded code does not automatically parallelize over the available cores. Algorithms other than embarrassingly parallel must be rethought, code must be rewritten with parallelism in mind, extreme care must be given to interference between threads when sharing memory, and, more generally, developers must learn new skills and rewrite their software to benefit from the multiple cores. It is often said that the exhaustion of Moore's law terminated a free option for developers. Modern hardware comes with better parallelism, but to benefit from that takes work. This part teaches readers the necessary skills to effectively program parallel algorithms in C++.

PARALLEL COMPUTING IN FINANCE

Given such progress in the development of parallel hardware and software, programmers from many fields have been writing parallel code for years. However, this trend only reached finance in the very recent years. Danske Bank, an institution generally well known for its cutting-edge technology, only implemented concurrent Monte-Carlo simulations in their production systems in 2013. This is all the more surprising as the main algorithm in finance, Monte-Carlo simulations, is embarrassingly parallel. The reason is there was little motivation for

parallel computing in investment banks prior to 2008–2011. Quantitative analysts almost exclusively worked on the valuation and risk management of exotics. Traders held thousands of exotic transactions in their books. Risk sensitivities were computed by "bumping" market variables one by one, then recomputing values. Values and risk sensitivities were additive across transactions. Hence, to produce the value and risk for a portfolio, the valuation of the transactions was repeated for every transaction and in every "bumped" market scenario. A large number of small valuations were conducted, which encouraged high-level parallelism, where valuations were distributed across transactions and scenarios but the internal algorithms that produce a value in a given context remained sequential.

That changed with regulatory calculations like CVA (Counterparty Value Adjustment) that estimates the loss incurred from a counterparty defaulting when the net present value (PV) of all transactions against this counterparty is positive. CVA is not additive due to netting effects across different transactions, and must be computed for all transactions at once, in one (particularly heavy) simulation. A CVA is also typically sensitive to thousands of market variables, and risk sensitivities cannot be produced by bumping in reasonable time. The response to this particular challenge, and arguably the strongest addition to computational finance over the past decade, is AAD, an alternative to bumping that computes all derivatives together with the value in constant time (explained in detail in Part III). So we no longer conduct many light computations but a single, extremely heavy one. The time taken by that computation is of crucial importance to financial institutions: slow computations result in substantial hardware costs and the inability to compute risk sensitivities in time. The computations must be conducted in parallel to take advantage of modern hardware. Hence, parallel computing only recently became a key skill for quantitative analysts and a central feature of modern financial libraries. It is the purpose of this part to teach these skills.

We cover concurrent programming under the Standard Threading Library, discuss the challenges involved, and explain how to develop effective parallel algorithms in modern C++. C++11 is a major modernization of C++ and includes a plethora of new features, constructs, and patterns. The Standard Threading Library is probably the most significant one, and it was designed consistently with the rest of C++11. For this reason, we review the main innovations of C++11 before we delve into its threading library.

Chapter 1 discusses high-performance programming in general terms, shows some concrete examples of the notions we just introduced, and delivers some important considerations for the development of fast applications in C++ on modern hardware. Chapter 2 discusses the most useful innovations in C++11, and, in particular, explains some important modern C++ idioms and patterns applied, among other places, in the Standard Threading Library. These patterns are also useful on their own right. Chapter 3 explores many aspects of concurrent programming with the Standard Threading Library, shows how to manage threads and send tasks for parallel execution, and illustrates its purpose with simple parallel algorithms. We also build a *thread pool*, which is applied in Parts II and III to situations of practical relevance in finance.

NOTES

1 We call such a function “thread safe.” It is a fundamental principle of functional programming that functions should not be allowed to have side effects. This is enforced in pure functional languages like Haskell, where all functions are thread safe and suitable for parallel application. C++ allows side effects, so thread safe design is the developer's responsibility. In particular, it should be made very clear which functions have side effects and which don't. Thread safe design is closely related to const correctness, as we will see in the next chapters.

2 To implement a binary decomposition is a trivial exercise, and an unnecessary one, since the chip's native representation of (unsigned) integers is binary.

3 A number of frameworks exist to facilitate distributed programming, the best known being the Message Passing Interface (MPI).

4 In our opinion, AAD is the most effective algorithm implemented in finance since FDM.

5 Although they may interfere through shared memory and cache as we will see later in this part.

6 Special instructions inserted in the body of the code that are not part of the C++ code to be compiled, but some directives for the compiler about how code is to be compiled. Pragmas are therefore compiler specific.

7 This result is known as Amdahl's law. It is immediately visible that, when the serial calculation time is Δ, the parallel calculation time is $\Delta[\mu/p + (1 - mu)]$, and the result follows.

8 The ratio of the speed-up over the number of parallel units is called “parallel efficiency.” Parallel efficiency is always between 0 (excluded) and 1 (included). The purpose of parallel algorithms is to achieve an efficiency as close as possible to 1.

9 Before Boost.Thread, the pthread C library offered similar functionality in plain C.