Preface

This book proposes a pedagogical introduction to shared memory application programming. It grew out of several years of user training at IDRIS supercomputing center in France, and, more recently, at the PATC training program of the PRACE European infrastructure. It is designed to guide readers with a reasonable background in C-C++ programming, in becoming acquainted with current multicore programming environments, and in developing useful insights about threads. This book is therefore placed at an intermediate level; basic experience in multithreaded programming is of course welcomed, but not mandatory.

Multithreaded programming is today a core technology, at the basis of any software development project in any branch of applied computer science. There are, naturally, a number of excellent presentations of the subject. We tried to provide in this book a basic overview of the multithreading landscape, so that readers can start mastering threads, benefiting from the abundant references that push these subjects much further in several directions. It seems therefore appropriate to expose in some detail the pedagogical motivations and strategies adopted here.

PEDAGOGICAL OBJECTIVES

Software engineering practices have experienced a profound evolution in the last 25 years, since shared memory programming first showed up. From simple, monolithic applications designed to solve a specific problem, dealing with a simple parallel context, we moved toward software packages dealing with complex, multicomponent modules, sometimes running in distributed computational environments. Another major evolution in software engineering occurred in the mid-2000s when working processor frequencies—and the peak performance that comes with it—ceased to double every 18 months, because a power dissipation wall was hit. Today, the only way to increase performance is to increase the number of processors, and for this reason multithreading, which was initially a question of choice and opportunity, has become a mandatory software technology. Threads are definitely a part of the life of any software developer concerned with issues of efficient network connectivity and I/O, interoperability, computational performance and scalability, or efficient interactive access to an application.

In this diversified and rapidly evolving landscape, a rather broad view of the utilities and the strategic options proposed by the different multithreading libraries and environments seems to be an important asset for software developers. An overview of different libraries and programming environments operating on Unix-Linux or Windows platforms is proposed, underlining their often complementary capabilities, as well as the fact that there is sometimes benefit in taking advantage of their interoperability. The first chapters of this book develop an overview of the native multithreading libraries like Pthreads (for Unix-Linux systems) or Windows threads, as well as the recent C++11 threads standard, with the purpose of grasping the fundamental issues on thread management and synchronization discussed in the following chapters. A quick introduction to the most basic OpenMP thread management interfaces is also introduced at this stage. In the second part of the book, a detailed discussion of the two high-level, standard programming environments—OpenMP and Intel Threading Building Blocks (TBB)—is given.

OpenMP—which played a fundamental role in disseminating shared memory programming—is today a broadly adopted and well-established standard. Nevertheless, we believe that a more general presentation of the basic concepts of shared memory programming, showing the way they are declined in the various programming environments, is useful for learning to think creatively about threads, not about programming languages. True enough, OpenMP can cope with practically any parallel context in an application. But the implementations of the basic multithreading concepts by different libraries often incorporate additional features that enable, in some specific cases, a better match to the application requirements. Some libraries propose specific extended utilities. A few cases will be given in which C++11 or TBB utilities are profitably used to boost application performance. This does not mean, however, that software developers need to abandon their well-rooted habits and best practices, because most of the time different libraries coexist peacefully and inter-operate inside an application. Examples are proposed of utilities developed using native or basic libraries, but exploited, for example, in an OpenMP environment.

This book focuses on parallel and concurrent patterns and issues that commonly occur in real applications, by discussing useful programming idioms, potential pitfalls, and preferred best practices that are very often independent of the underlying programming environment. In any subject, however, complex, there is always a limited number of important concepts and ideas to be solidly grasped. Once they are mastered, the rest is information that can be easily integrated when needed. Being able to think creatively about threads is more important than knowing every possible detail about specific libraries. In training users, our purpose has always been to enhance their insights on threads and to broaden their understanding of the strategic design choices open to them. For this purpose, a substantial number of examples are proposed in each chapter, with commented sources. All, without exception, are applications of varying degrees of complexity that can be compiled, run, and modified if needed. Many examples involve input parameters that can be changed to better understand the code behavior. A special effort has been made to propose simple but realistic examples, dealing with common issues in application programming.

PROGRAMMING ENVIRONMENTS

This book is addressed to C-C++ programmers. Deep expertise in C++ is not required. True enough, a major step today is the increasing availability of a standard C++11 thread library, but we tried to avoid excluding C programmers. Two of the programming environments discussed here (C++11 and TBB) are C++ libraries, but most of the examples and applications only rely on the usage of C++ objects, an issue easily handled by C programmers. Some more advanced C++ features used by TBB or C++11—like function objects and lambda expressions—are discussed in annex B.

The role, scope, and potential usefulness of the different programming environments discussed in this book will be clarified as we go, but it is useful to take a first look at their relevance. We will henceforth call *native libraries* the multithreaded support provided by different operating systems: the Pthreads (Posix Threads) library for Unix-Linux systems and the Windows API (formerly called Win32 API) for Windows. These native libraries provide the primitives needed to construct multithreaded applications, as well as a number of useful utilities.

The C++11 standard has incorporated a complete set of multithreading facilities into the C++ Standard Library. This evolution was motivated by the obvious interest in boosting C++ to a thread

aware programming language, to provide C++ programmers with the capability of disposing of a portable and efficient environment for refined code development. Performance was a basic requirement: the C++11 thread library is designed to support sophisticated system programming without the help of the more basic native programming interfaces. It provides the same basic services as the native libraries mentioned before, as well as a few other refinements and extensions aiming at facilitating code optimization. Given the increasing availability of fully C++11 compliant compilers—like GNU 4.8 or higher in Linux, and Visual Studio 2013 in Windows—we have incorporated C++11 threads in this book on the same footing as Pthreads or Windows threads.

Boost is a prestigious, portable, general-purpose C++ library providing powerful utilities in all areas of C++ software development. It includes a multithreading library fully compliant with the C++11 standard. In fact, Boost was the testing ground where the C++11 standard was developed. We will occasionally refer to some specific facilities proposed by Boost.

OpenMP and TBB are programming environments proposing high-level programming interfaces encapsulating the native libraries services or the very low-level operating system support for multithreading, adding value and features. These environments are portable in the sense discussed above: the same programming interfaces with different implementations in different operating systems. The fundamental added value with respect to native libraries is the availability of high-level interfaces for thread management, synchronization, and scheduling, simplifying the programming and deployment of parallel applications. It would not be appropriate, however, to completely disregard the basic libraries on the basis that they are low-level environments. True enough, it takes a substantial amount of time and effort to implement a complex application with only the basic libraries. But they have indeed many features that are definitely not low level, easily accessible to application programmers. And, on the other hand, there are programming contexts that require subtle low-level programming even in a high-level environment like OpenMP, simply because the high-level tools required to cope with them are not proposed by the environment. This is the reason why we believe it is useful and instructive to keep a critical eye on the basic multithreading libraries.

There are other useful and efficient programming environments that are not discussed in this book, for portability reasons. OpenMP requires compiler support, but it is automatically integrated in any C-C++ compiler. TBB, instead, is a portable library not requiring compiler support. These environments are therefore fully portable. Another very attractive environment is Intel CilkPlus [1], with easy-to-use C-C++ programming interfaces, based on software technologies also adopted by TBB. CilkPlus, however, requires compiler support, traditionally provided by the Intel C-C++ compiler, which limits its portability. But the expected support in the GNU compiler may change the situation in the future. Last but not least, Microsoft has traditionally provided very strong multithreaded support for Windows, today implemented in a set of Concurrency Runtime tools, incorporating, among other things, the Microsoft Parallel Patterns Library (PPL) [2]. A close collaboration between Microsoft and Intel has established a large amount of compatibility between PPL and TBB.

Besides the basic libraries and high-level programming environments mentioned above, this book relies on another relatively small, high-level library we developed during the last few years, called vath. This library proposes some high-level, easy-to-use utilities in the form of C++ classes that encapsulate the low-level primitives provided by the basic libraries. Our initial motivation was to simplify the usage of the thread management utilities proposed by the native and basic libraries and to dispose of simple tools to cope with specific parallel contexts not easily handled in OpenMP. The specific added value that this additional library may provide will be discussed in later chapters. The point we want

to make here is that it is as portable as OpenMP or TBB: the same programming interfaces are implemented in Pthreads (for Unix-Linux systems) and in Windows threads. And there is also a third implementation using the C++11 thread library. This book focuses on the use of this library, not on its implementation. However, for pedagogical reasons, a maximal simplicity programming style has been adopted in the source codes, in order to make it easily accessible to readers interested in taking a closer look at basic Pthreads, Windows of C++11 programming.

As far as the application examples are concerned, with the exception of a few cases in which our purpose is to show some specific feature of one of the basic libraries, we access Pthreads or Windows threads via portable code using the vath library. Most of the code examples have a vath version and an OpenMP version and, whenever relevant, a TBB version.

BOOK ORGANIZATION

Chapter 1 reviews the basic concepts and features of current computing platforms needed to set the stage for concurrent and parallel programming. Particular attention is paid to the impact of the multicore evolution of computing architectures, as well as other basic issues having an impact on efficient software development, such as the hierarchical memory organization, or the increased impact of rapidly evolving heterogeneous architectures incorporating specialized computing platforms like Intel Xeon Phi and GPU accelerators.

Chapter 2 introduces a number of basic concepts and facts on multithreading processing that are totally generic and independent of the different programming environments. They summarize the very general features that determine the way multithreading is implemented in existing computing platforms.

Chapter 3 introduces threads in detail. First, the programming interfaces proposed by Pthreads, Windows, and C++11 to create and run a team of worker threads are discussed, with the help of specific examples. Then, a portable thread management utility from the vath library is introduced, encapsulating the basic programming interfaces. Finally, a first look at OpenMP is taken to show how the same thread management features are declined in this environment. A few examples introduce some basic multithreading issues, like thread safety or thread synchronization. This initial part of the book adopts the traditional *thread-centric programming style* in which the activity of each thread in the application is perfectly identified.

Chapter 4 examines the thread safety issue for function calls: under what conditions a library function provides the correct, expected service when asynchronously called by several threads in a multithreaded environment? Random number generators are used as examples of functions that are not thread-safe. A detailed discussion follows, concerning the thread-specific storage tools proposed by different programming environments (OpenMP, TBB, C++11, Windows) in order to enforce thread safety in functions that are not by nature thread-safe. Best practices concerning the thread-specific storage tools are discussed.

Chapters 5 and 6 deal with the fundamental issue of thread synchronization. Two basic thread synchronization primitives – mutual exclusion and event synchronization – are introduced. Mutual exclusion deals with thread safety in shared data accesses, and in its simplest forms this primitive is easy to use in application programming. Chapter 5 examines in detail the mutual exclusion interfaces available in the five programming environments, underlining the different additional features proposed by each one of them. Then, atomic operations are introduced as an efficient alternative to mutual

exclusion in some specific cases, and the way they operate in OpenMP is reviewed (an in-depth discussion of atomic operations is reserved to Chapter 8). Finally, the concurrent container classes provided by TBB are described. They correspond to extensions of the Standard Template Library containers having internally built-in thread safety, not requiring explicit mutual exclusion in their data accesses. This chapter concludes with a number of observations concerning mutual exclusion pitfalls and best practices.

Chapter 6 is concerned with event synchronization, namely, how to force a thread to wait for an event triggered by another thread (an event can be, for example, a change of value of some shared data item). A detailed pedagogical discussion of the event synchronization primitives in Pthreads, Windows, and C++11 is presented. However, direct use of these event synchronization primitives is indeed low-level programming, and they are never used as such in the rest of the book. They will show up encapsulated in higher level, easy-to-use vath utilities discussed in Chapter 9.

Chapter 7 focuses on the fundamental problem in shared memory programming: given the asynchronous behavior of threads and the hierarchical structure of the memory system, how can we know for sure that a shared data access retrieves the correct, expected data value? This problem leads to the notion of memory consistency model, and a simple, high-level discussion of the relevance of this concept is presented. This leads to the fundamental conclusion of this chapter: mutual exclusion, initially introduced to enforce thread safety in memory write operations, is also required to enforce memory consistency in memory read operations.

On the basis of the previous memory consistency concepts, Chapter 8 presents a rather detailed discussion of the powerful atomic classes proposed by C++11 and TBB, as well as the Windows atomic services. Besides the obvious use in replacing mutex locking, this chapter discusses the way they can be used to implement efficient, custom synchronization utilities. First, the discussion focuses on the way to enforce thread safety in shared data accesses, not using mutual exclusion: the so-called lock-free algorithms. Next, it is shown how the memory consistency constraints embedded in the atomic services are used to implement custom synchronization patterns based on "happens before" relations among operations in different threads. Two examples are proposed, implementing synchronization utilities incorporated in the vath library.

Chapter 9 presents an overview of high-level thread synchronization utilities incorporated in the vath library. They implement synchronization patterns common in applications programming (synchronized memory read-writes, Boolean locks, blocking barriers, reader-writer locks, concurrent queues) that are not immediately available in OpenMP. A pure OpenMP version of some these utilities is also provided. The discussion on the scope and impact of these utilities, as well as the examples proposed, is designed to significantly contribute to sharpen the understanding of thread synchronization pitfalls and best practices.

Chapters 10 and 11 are the core of this book: the presentation of the OpenMP and Intel TBB programming environments. The focus moves at this point to thread pool interfaces—the underlying architecture of both programming environments—where threads are created first and wait silently in a blocked state to be activated to execute a task. The traditional OpenMP programming model relies on a thread-centric pool, in the sense that there is a clear mapping between parallel tasks and threads, and it is possible to know at any time what every thread is doing. If more parallelism in needed in the applications, more threads are added to the pool. Nearly 10 years ago, an alternative programming model emerged, pioneered by Cilk [3], in which any number of parallel tasks are executed by a fixed number of threads, their execution being organized in an innovative way with a substantial amount of

intelligence in the task scheduler itself. In such a task-centric environment, it is not possible to know what every individual thread is doing at a given time. TBB first implemented this model as a standalone library, and at about the same time OpenMP started to integrate task-centric software technologies in the 3.0 and 3.1 versions. The recent 4.0 version also makes a substantial step in this direction. Therefore, at this point readers are led to move progressively from thinking about threads to thinking about parallel tasks. The focus starts to move from thread centric to task-centric programming, particularly adapted to irregular, recursive, or unbalanced parallel contexts.

Chapter 10 presents a complete overview of OpenMP: the traditional programming model as well as a full discussion and examples of the new features incorporated in the latest OpenMP 4.0 release. Particular attention is given to the pitfalls and best practices related to the task directive. Besides the 4.0 extension of the task API incorporating a substantial number of additional task-centric features, the other major 4.0 extensions are also reviewed, like the cancellation of parallel constructs, thread affinity, and directives for vectorization and code offloading to external accelerators.

Chapter 11 concentrates on the TBB programming environment. TBB proposes a number of useful standalone utilities, already discussed in previous chapters. This chapter concentrates on the high-level TBB programming environment, based on a number of easy-to-use, STL like automatic parallelization algorithms that cope with a variety of different parallel patterns. Then, the discussion moves to a recent programming interface providing a simplified access to the most relevant features of the task scheduler, and several of the examples proposed in the OpenMP chapter are reexamined in this context. The discussion of the full task scheduler API is delayed to Chapter 16.

Chapter 12 discusses yet another thread management utilities: the SPool and NPool thread pool classes proposed by the vath library, implementing, respectively, a thread-centric or a task-centric programming environment. The SPool class has already been used to implement simple parallel patterns in previous chapters. Our motivation in developing these thread pools was not to compete with professional, well-established programming standards, but to benefit from some additional programming comfort in some specific cases not easily handled by OpenMP. Unlike OpenMP or TBB, thread pools are explicitly created by the user. An application can dispose of several independent pools, and parallel jobs can be submitted to all of them. Client threads that submit jobs are always external to the pools, but they are correctly synchronized with the jobs they submit. The task scheduler of the NPool utility is less refined than those implemented in OpenMP or TBB, but the pool can run most of the OpenMP or TBB examples proposed in the two previous chapters. These environment makes it very easy to run several parallel routines in parallel, and an explicit example of the added value they provide is presented at the end of the chapter.

The three following chapters discuss a few full applications taken from computational sciences, each one concentrating on one specific parallel context. The purpose here is to clearly understand the multithreaded architecture of the applications and the thread synchronization requirements that follow. We also look in detail at their scaling properties and at the impact on performance of excessive synchronization overhead. It is here that an example is met in which mediocre scaling properties are improved by the use of custom synchronization tools constructed with TBB or C++11 utilities. Chapter 13 discusses a molecular dynamics application—computing the trajectories of a set of interacting particles—exhibiting a substantial demand of barrier synchronization. Chapter 14 deals with data parallelism, developing two classical examples of domain decomposition in solving partial differential equations. Chapter 15 discusses pipelining, a kind of control parallelism totally different from data parallelism. Two complete pipelining applications are developed, one of them dealing with a image

treatment problem. In all cases, vath, OpenMP and TBB versions of the code are proposed (which are sometimes very similar).

> The examples in Chapters 13–15 deal with simple but realistic scientific problems. However, a significant pedagogical effort has been made to make them accessible to readers not having a scientific background. Simple descriptions are provided of the relevance and interest of the problem, but the discussion concentrates on the solutions to the programming challenges raised by the application.

Last but not least, Chapter 16 deals with the programming interfaces enabling direct access to the TBB task scheduler. This subject, admittedly a rather advanced subject on task parallelism, is nevertheless interesting and relevant. It was often emphasized in preceding chapters that the event synchronization primitives and high-level utilities discussed in Chapters 6 and 8 are not adapted to be used in task-centric environments (TBB, NPool, or the OpenMP task API), because they are programmed inside parallel tasks, but they really synchronize the underlying executing threads. This is fine as long as a controlled, one-to-one mapping of tasks to threads is established. However, in task-centric environments this issue is handled by introducing specific task synchronization features in the task scheduler itself, enabling programmers to synchronize tasks, not threads. Chapter 10 covers a OpenMP 4.0 extension, based on the new depends directive, that goes in this direction. The TBB task scheduler API implements task synchronization by using advanced scheduler features to organize the way the parallel tasks are scheduled and executed. In Chapter 16, a number of task synchronization examples are provided. In addition, the task scheduler is used to explore possible optimization strategies designed to improve the performance of the molecular dynamics application discussed in Chapter 13.

Finally, a few more words about support software, which can be downloaded from the book site http://booksite.elsevier.com/9780128037614. There are, in early chapters, some examples that target specific Pthreads, Windows, or C++11 features. But the majority of examples access the basic library utilities through the portable vath library presented in Chapters 9 and 12. The examples are therefore fully portable, and it is up to the reader to decide what implementation to use. Annex A provides a complete description of the accompanying software, as well as detailed instructions for installation of the different software components, and for compilation and execution of the vath library and the code examples.