Preface

All computers are now parallel computers, so we assert that all programmers are, or should be, parallel programmers. With parallel programming now mainstream, it simply needs to be included in the definition of "programming" and be part of the skill set of *all* software developers. Unfortunately, many existing texts on parallel programming are overspecialized, with too much emphasis given to particular programming models or particular computer architectures. At the other extreme, several existing texts approach parallel computing as an abstract field of study and provide the reader with insufficient information to actually write real applications. We saw a need for a text on parallel programming treating the topic in a mainstream, pragmatic fashion, so that readers can immediately use it to write real applications, but at a level of abstraction that still spans multiple computer architectures and programming models.

We feel that teaching parallel programming as an advanced topic, with serial programming as the default basis, is not the best approach. Parallel programming should be taught from the beginning to avoid over-learning of serial assumptions and thought patterns. Unfortunately, at present the default teaching style is based on the serial code and algorithms. Serialization has become excessively woven into our teaching, our programs, and even the tools of our trade: our programming languages. As a result, for many programmers parallel programming seems more difficult than it should be. Many programs are serial not because it was natural to solve the problem serially, but because the programming tools demanded it and the programmer was trained to think that way.

Despite the fact that computer hardware is naturally parallel, computer architects chose 40 years ago to present a serial programming abstraction to programmers. Decades of work in computer architecture have focused on maintaining the illusion of serial execution. Extensive efforts are made inside modern processors to translate serial programs into a parallel form so they can execute efficiently using the fine-grained parallel hardware inside the processor. Unfortunately, driven by the exponential increase in the number of transistors provided by Moore's Law, the need for parallelism is now so great that it is no longer possible to maintain the serial illusion while continuing to scale performance. It is now necessary for programmers to *explicitly* specify parallel algorithms if they want their performance to scale. Parallelism is everywhere, and it is the path to performance on modern computer architectures. Parallelism, even within a single desktop (or laptop!) computer, is available in many ways, including vector (SIMD) instructions, multicore processors, GPUs, co-processors, and many-core processors. Programming today needs to address all these forms of hardware parallelism in a manner that is abstract enough to avoid limiting the implementation to a particular style of hardware.

We also saw a need for a structured approach to parallel programming. In this book, we explain and illustrate essential strategies needed for writing efficient, scalable programs using a set of *patterns*. We have found that patterns are highly effective both for learning this subject and for designing efficient, structured, and maintainable programs. Using standard names for patterns is also a tremendous aid to communication. Because vocabulary is important, we have assembled an extensive glossary. This should help limit the need to read the book sequentially. The glossary also points to key discussions or explanations of a term within a section of the book when appropriate.

To ensure that the book is useful in practice, we combine patterns with a set of examples showing their use. Since there are many parallel programming models, the question arose: Which programming model(s) should we use for examples? We wanted to show enough examples to allow the reader to write sophisticated applications without having to depend heavily on external references. That constraint argued for sticking to one programming model or a small number of them. On the other hand, we wanted to demonstrate that the patterns we are presenting are universal and span a large number of programming models.

As a compromise, we decided to show a large number of examples focused on a couple of primary models and a small number in other "secondary" models. For the primary models, we chose Intel Threading Building Blocks (Intel TBB) and Intel Cilk Plus. These two models are efficient and well-supported. Both are readily available, with both open source licenses and commercial support. TBB is a C++ template library that works with many different ISO C++ compilers, while Cilk Plus is a C/C++ language extension, so they provide contrasting syntactic approaches. Together they are capable of expressing all the patterns discussed in this book. Complete working code for all the examples in the primary programming models, as well as a variety of other material, can be found online at

http://parallelbook.com

We feel a sufficient number of examples have been provided that, along with the standard documentation, this book can be used for learning how to program in both TBB and Cilk Plus.

However, the patterns we discuss apply to almost any parallel programming model; therefore, to provide a broader perspective, we look at three secondary programming models: Intel Array Building Blocks (ArBB), OpenCL, and OpenMP. ArBB uses a parallel virtual machine provided as a library. The ArBB virtual machine (VM) supports explicit, programmer-directed runtime code generation and is designed to be usable from multiple languages. In this book, we use the C++ front-end to the ArBB VM, which embeds a parallel language syntax into C++ using normal C++ mechanisms such as macros and operator overloading. We also show some examples in OpenCL and OpenMP. Both OpenCL and OpenMP are standards, with OpenCL primarily designed to target GPU-like architectures, and OpenMP targeting shared-memory multicore CPU architectures. OpenCL is based on a separately compiled kernel language which is provided as a string to an library interface. Like ArBB, OpenCL supports dynamic compilation. In contrast, OpenMP is based on annotations in an existing language and is designed to be statically compiled. These five programming models take different syntactic approaches to the expression of parallelism, but as we will see, the patterns apply to all of them. This reinforces the fact that patterns are universal, and that a study of patterns is useful not only for today's programming models but also for what may come in the future.

This book is neither a theory book nor a cookbook. It is a pragmatic strategic guide, with case studies, that will allow you to understand how to implement efficient parallel applications. However, this book is not aimed at supercomputing programmers, although it might be interesting to them. This is a book for mainstream C and C++ programmers who may have no prior knowledge of parallel programming and who are interested in improving the performance of their applications. To this end, we also discuss performance models. In particular, we present the work-span model of parallel complexity, which goes beyond the simplistic assumptions of Amdahl's Law and allows better prediction of an algorithm's speedup potential, since it gives both upper and lower bounds on speedup and provides a tighter upper bound.

We hope to provide you with the capacity to design and implement efficient, reliable, and maintainable parallel programs for modern computers. This book purposely approaches parallel programming from a programmer's point of view without relying on an overly detailed examination or prior knowledge of parallel computer architecture. We have avoided the temptation to make this a computer architecture book. However, we have still taken care to discuss important architecture constraints and how to deal with them. The goal is to give you, as a programmer, precisely the understanding of the computer architecture that you need to make informed decisions about how to structure your applications for the best performance and scalability.

Our vision is that effective parallel programming can be learned by studying appropriate patterns and examples. We present such a set of patterns, give them concrete names, and ground them in reality with numerous examples. You should find that this approach directly enables you to produce effective and efficient parallel programs and also allows you to communicate your designs to others. Indirectly, it may spark interest in parallel computer architecture, parallel language design, and other related topics. No book could possibly include information on all the topics of interest related to parallel programming, so we have had to be selective. For those so inclined, we have included suggestions for further reading in an appendix. Our web site at http://parallelbook.com also includes material that goes beyond the basics presented here.

We hope you find this book to be effective at extending your definition of "programming" to include parallel programming.

James Reinders
Portland, Oregon, USA

Arch Robison
Champaign, Illinois, USA

Michael McCool
Waterloo, Ontario, Canada
Tokyo, Japan

In the short time since this book was first published, the use of parallelism to improve computing performance has continued to evolve rapidly. New parallel processors have become available and parallel programming models have been extended. With these recent developments the topics covered by this book are becoming increasingly relevant. In particular, the need for parallel programming and the patterns and programming techniques introduced in this book are becoming even more important.

In November 2012, the Intel Xeon Phi coprocessors using the MIC architecture referred to in this book were introduced by Intel, and have already been deployed in several supercomputers. The Intel Xeon Phi coprocessors have proven to be so efficient that supercomputers built using them have placed highly on the Green 500 list of energy-efficient supercomputers, including the Beacon supercomputer which occupies the #1 spot for power efficiency. Supercomputers with these coprocessors have also been used in 7 out of 10 computers on the Top 500 as of November 2012. The Intel Xeon Phi co-processor has up to 61 cores each supporting 512-bit-wide vector instructions, and is capable of over a trillion double-precision floating-point operations a second. Using the Xeon Phi coprocessor effectively requires a significant of parallelism. Its full potential can only be achieved with scalable parallel programs that are also highly vectorized. Writing such programs is, of course, the focus of this book. Fortunately, exactly the same programming models (and parallel patterns) that are useful for ordinary multicore processors can be used with the Xeon Phi coprocessor.

Parallelism is not just about supercomputers. The processors used in laptops and desktops continue to advance, with more cores and larger and more powerful integrated graphics engines. Most importantly, these new graphics engines feature close integration with the CPUs that they share die area and memory systems with. Discrete graphics processors have evolved as well, with the latest models incorporating more and more CPU-like features. In general, parallel programming models appear to be converging. For example, since this book was first released, an explicitly vectorizing loop construct was voted into the next OpenMP specification. This vectorizing construct is close to the ñpragma simdò construct that we introduce in this book under Cilk Plus, and so brings OpenMP and Cilk Plus into closer alignment. Cilk Plus tasking constructs have also been proposed for C++ standardization, although that proposal is still under discussion.

These changes have taken place in a mere eight months since the English version of this book was finalized. The computer industry is maintaining its rapid pace of innovation. Fortunately, this book was written based on patterns. Parallel patterns span multiple programming models and computer architectures, and the content has only grown in relevance. The only way to make effective use of a massively parallel architecture like the Intel Xeon Phi coprocessor is to use techniques like those described in this book. Moore's Law shows no sign of abating, and so we can expect to see such techniques become essential parts of programming. As all processors become massively parallel, programmers need to make parallel programming part of their ordinary practice in order to see continued increases in performance with each new generation.

We hope that this book will help you realize the potential of these processors even as they continue to evolve. We also hope that this book will help establish definitions and terminology for parallel patterns so that software developers can more clearly define, articulate, and implement high-performance parallel architectures for their applications.

Michael McCool
Arch Robison
James Reinders
January 2013