More than ever, multithreading is a requirement for good performance of systems with multi-core chips. This guide explains how to maximize the benefits of these processors through a portable C++ library that works on Windows, Linux, Macintosh, and Unix systems. With it, you'll learn how to use Intel Threading Building Blocks (TBB) effectively for parallel programming—without having to be a threading expert.

Written by James Reinders, Chief Evangelist of Intel Software Products, and based on the experience of Intel's developers and customers, this book explains the key tasks in multithreading and how to accomplish them with TBB in a portable and robust manner. With plenty of examples and full reference material, the book lays out common patterns of uses, reveals the gotchas in TBB, and gives important guidelines for choosing among alternatives in order to get the best performance.

Any C++ programmer who wants to write an application to run on a multi-core system will benefit from this book. TBB is also very approachable for a C programmer or a C++ programmer without much experience with templates. Best of all, you don't need experience with parallel programming or multi-core processors to use this book.
Praise for *Intel Threading Building Blocks*

“The Age of Serial Computing is over. With the advent of multi-core processors, parallel-computing technology that was once relegated to universities and research labs is now emerging as mainstream. Intel Threading Building Blocks updates and greatly expands the ‘work-stealing’ technology pioneered by the MIT Gilk system of 15 years ago, providing a modern industrial-strength C++ library for concurrent programming.

“Not only does this book offer an excellent introduction to the library, it furnishes novices and experts alike with a clear and accessible discussion of the complexities of concurrency.”

— Charles E. Leiserson, MIT Computer Science and Artificial Intelligence Laboratory

“We used to say make it right, then make it fast. We can’t do that anymore. TBB lets us design for correctness and speed up front for Maya. This book shows you how to extract the most benefit from using TBB in your code.”

— Martin Watt, Senior Software Engineer, Autodesk

“TBB promises to change how parallel programming is done in C++. This book will be extremely useful to any C++ programmer. With this book, James achieves two important goals:

• Presents an excellent introduction to parallel programming, illustrating the most common parallel programming patterns and the forces governing their use.
• Documents the Threading Building Blocks C++ library—a library that provides generic algorithms for these patterns.

“TBB incorporates many of the best ideas that researchers in object-oriented parallel computing developed in the last two decades.”

— Marc Snir, Head of the Computer Science Department, University of Illinois at Urbana-Champaign

“This book was my first introduction to Intel Threading Building Blocks. Thanks to the easy-to-follow discussion of the features implemented and the reasons behind the choices made, the book makes clear that Intel’s Threading Building Blocks are an excellent synthesis of some of the best current parallel programming ideas. The judicious choice of a small but powerful set of patterns and strategies makes the system easy to learn and use. I found the numerous code segments and complete parallel applications presented in the book of great help to understand the main features of the library and illustrate the different ways it can be used in the development of efficient parallel programs.”

— David Padua, University of Illinois
“The arrival of the multi-core chip architecture has brought great challenges in parallel programming and there is a tremendous need to have good books that help and guide the users to cope with such challenges.

“This book on Intel Threading Building Blocks provides an excellent solution in this direction and is likely to be an important text to teach its readers on parallel programming for multi-cores.

“The book illustrates a unique path for readers to follow in using a C++-based parallel programming paradigm—a powerful and practical approach for parallel programming. It is carefully designed and written, and can be used both as a textbook for classroom training, or a cookbook for field engineers.”

—Professor Guang R. Gao, University of Delaware

“I enjoyed reading this book. It addresses the need for new ways for software developers to create the new generation of parallel programs. In the presence of one of the ‘largest disruptions that information technology has seen’ (referring to the introduction of multicore architectures), this was desperately needed.

“This book also fills an important need for instructional material, educating software engineers of the new opportunities and challenges.

“The library-based approach, taken by the Threading Building Blocks, could be a significant new step, as it complements approaches that rely on advanced compiler technology.”

—Rudolf Eigenmann, Purdue University, Professor of ECE and Interim Director of Computing Research Institute

“Multi-core systems have arrived. Parallel programming models are needed to enable the creation of programs that exploit them. A good deal of education is needed to help sequential programmers adapt to the requirements of this new technology. This book represents progress on both of these fronts.

“Threading Building Blocks (TBB) is a flexible, library-based approach to constructing parallel programs that can interoperate with other programming solutions.

“This book introduces TBB in a manner that makes it fun and easy to read. Moreover, it is packed full of information that will help beginners as well as experienced parallel programmers to apply TBB to their own programming problems.”

—Barbara Chapman, CEO of cOMPunity, Professor of Computer Science at the University of Houston
“Future generations of chips will provide dozens or even hundreds of cores. Writing applications that benefit from the massive computational power offered by these chips is not going to be an easy task for mainstream programmers who are used to sequential algorithms rather than parallel ones.

“Intel’s TBB is providing a big step forward into this long path, and what is better, all in the C++ framework.”

—Eduard Ayguade, Barcelona Supercomputer Center, Technical University of Catalunya

“Intel’s TBB is to parallel programming what STL was to plain C++. Generic programming with STL dramatically improved C++ programming productivity. TBB offers a generic parallel programming model that hides the complexity of concurrency control. It lowers the barrier to parallel code development, enabling efficient use of ‘killer’ multi-cores.”

—Lawrence Rauchwerger, Texas A&M University, Inventor of STAPL

“For the last eighteen years the denizens of the thinly populated world of supercomputers have been looking for a way to write really pretty and practical parallel programs in C++. We knew templates and generic programming had to be part of the answer, but it took the arrival of multi-core (and soon many-core) processors to create a fundamental change in the computing landscape. Parallelism is now going to be everyday stuff.

“Every C++ programmer is going to need to think about concurrency and parallelism and Threading Building Blocks provides the right abstractions for them to do it correctly.

“This book is not just a discussion of a C++ template library. It provides a lovely and in-depth overview of much of what we have learned about parallel computing in the last 25 years. It could be a great textbook for a course on parallel programming.”

—Dennis Gannon, Science Director, Pervasive Technology Labs at Indiana University, former head of DARPA’s High Performance Computing (HPC++) project, and steering committee member of the Global Grid Forum

“TBB hits the application developer’s sweet spot with such advantages as uniprocessor performance, parallel scalability, C++ programming well beyond OpenMP, compatibility with OpenMP and hand threads, Intel Threading Tools support for performance and confidence, and openness to the software development community. TBB avoids several constraints surrounding the sweet spot: language extension risks, specific compiler dependences and hand-threading complexities.

“This book should make developers productive without a steep training curve, and the applications they produce should be of high quality and performance.”

—David Kuck, Intel Fellow, founder of KAI and former director of the Center for Supercomputing Research and Development
# Table of Contents

**Foreword** .................................................................................................................. xiii

**Note from the Lead Developer of Intel Threading Building Blocks** ....................... xv

**Preface** ......................................................................................................................... xix

1. **Why Threading Building Blocks?** ........................................................................... 1  
   Overview .................................................................................................................. 2  
   Benefits ................................................................................................................... 2

2. **Thinking Parallel** .................................................................................................. 7  
   Elements of Thinking Parallel ................................................................................. 8  
   Decomposition ....................................................................................................... 9  
   Scaling and Speedup ............................................................................................. 13  
   What Is a Thread? .................................................................................................... 19  
   Mutual Exclusion and Locks .................................................................................. 22  
   Correctness ............................................................................................................ 23  
   Abstraction ............................................................................................................ 25  
   Patterns .................................................................................................................. 25  
   Intuition .................................................................................................................. 27

3. **Basic Algorithms** .................................................................................................. 29  
   Initializing and Terminating the Library .................................................................. 30  
   Loop Parallelization ............................................................................................... 32  
   Recursive Range Specifications ............................................................................ 52  
   Summary of Loops ................................................................................................. 64
4. **Advanced Algorithms** ................................................................. 65
   Parallel Algorithms for Streams ............................................... 65

5. **Containers** ................................................................. 80
   concurrent_queue ................................................................. 81
   concurrent_vector ............................................................... 86
   concurrent_hash_map ........................................................... 91

6. **Scalable Memory Allocation** ............................................... 101
   Limitations ............................................................................. 101
   Problems in Memory Allocation ............................................. 101
   Memory Allocators ............................................................... 103
   Replacing malloc, new, and delete ......................................... 104

7. **Mutual Exclusion** ............................................................... 110
   When to Use Mutual Exclusion .............................................. 111
   Mutexes ................................................................................. 112
   Mutexes ................................................................................. 118
   Atomic Operations .................................................................. 122

8. **Timing** ............................................................................. 130

9. **Task Scheduler** ................................................................. 133
   When Task-Based Programming Is Inappropriate .................. 133
   Much Better Than Raw Native Threads ................................ 134
   Initializing the Library Is Your Job ....................................... 137
   Example Program for Fibonacci Numbers ......................... 137
   Task Scheduling Overview .................................................... 140
   How Task Scheduling Works ............................................... 142
   Recommended Task Recurrence Patterns ............................ 145
   Making Best Use of the Scheduler ....................................... 147
   Task Scheduler Interfaces .................................................... 153
   Task Scheduler Summary ...................................................... 168

10. **Keys to Success** ................................................................. 169
    Key Steps to Success .......................................................... 169
    Relaxed Sequential Execution ............................................ 170
    Safe Concurrency for Methods and Libraries ..................... 171
Debug Versus Release 172
For Efficiency’s Sake 172
Enabling Debugging Features 172
Mixing with Other Threading Packages 174
Naming Conventions 176

11. Examples ................................................................. 177
   The Aha! Factor 177
   A Few Other Key Points 179
   parallel_for Examples 180
   The Game of Life 190
   Parallel_reduce Examples 199
   CountStrings: Using concurrent_hash_map 209
   Quicksort: Visualizing Task Stealing 215
   A Better Matrix Multiply (Strassen) 223
   Advanced Task Programming 230
   Packet Processing Pipeline 237
   Memory Allocation 257
   Game Threading Example 262
   Physics Interaction and Update Code 271
   Open Dynamics Engine 275

12. History and Related Projects ................................. 283
   Libraries 283
   Languages 285
   Pragmas 286
   Generic Programming 286
   Caches 289
   Costs of Time Slicing 290
   Quick Introduction to Lambda Functions 291
   Further Reading 292

Index ............................................................................. 297
This excerpt is protected by copyright law. It is your responsibility to obtain permissions necessary for any proposed use of this material. Please direct your inquiries to permissions@oreilly.com.
Chapter 1

Why Threading Building Blocks?

Intel Threading Building Blocks offers a rich and complete approach to expressing parallelism in a C++ program. It is a library that helps you leverage multi-core processor performance without having to be a threading expert. Threading Building Blocks is not just a threads-replacement library; it represents a higher-level, task-based parallelism that abstracts platform details and threading mechanisms for performance and scalability.

This chapter introduces Intel Threading Building Blocks and how it stands out relative to other options for C++ programmers. Although Threading Building Blocks relies on templates and the C++ concept of generic programming, this book does not require any prior experience with these concepts or with threading.

Chapter 2 explains the challenges of parallelism and introduces key concepts that are important for using Threading Building Blocks. Together, these first two chapters set up the foundation of knowledge needed to make the best use of Threading Building Blocks.

Download and Installation


Threading Building Blocks was initially released in August 2006 by Intel, with prebuilt binaries for Windows, Linux, and Mac OS X. Less than a year later, Intel provided more ports and is now working with the community to provide additional ports. The information on how to install Threading Building Blocks comes with the product downloads.
Overview

Multi-core processors are becoming common, yet writing even a simple `parallel_for` loop is tedious with existing threading packages. Writing an efficient scalable program is much harder. Scalability embodies the concept that a program should see benefits in performance as the number of processor cores increases.

Threading Building Blocks helps you create applications that reap the benefits of new processors with more and more cores as they become available.

Threading Building Blocks is a library that supports scalable parallel programming using standard C++ code. It does not require special languages or compilers. The ability to use Threading Building Blocks on virtually any processor or any operating system with any C++ compiler makes it very appealing.

Threading Building Blocks uses templates for common parallel iteration patterns, enabling programmers to attain increased speed from multiple processor cores without having to be experts in synchronization, load balancing, and cache optimization. Programs using Threading Building Blocks will run on systems with a single processor core, as well as on systems with multiple processor cores. Threading Building Blocks promotes scalable data parallel programming. Additionally, it fully supports nested parallelism, so you can build larger parallel components from smaller parallel components easily. To use the library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner. The result is that Threading Building Blocks enables you to specify parallelism far more conveniently, and with better results, than using raw threads.

Benefits

As mentioned, the goal of a programmer in a modern computing environment is scalability: to take advantage of both cores on a dual-core processor, all four cores on a quad-core processor, and so on. Threading Building Blocks makes writing scalable applications much easier than it is with traditional threading packages.

There are a variety of approaches to parallel programming, ranging from the use of platform-dependent threading primitives to exotic new languages. The advantage of Threading Building Blocks is that it works at a higher level than raw threads, yet does not require exotic languages or compilers. You can use it with any compiler supporting ISO C++. This library differs from typical threading packages in these ways:

*Threading Building Blocks enables you to specify tasks instead of threads*

Most threading packages require you to create, join, and manage threads. Programming directly in terms of threads can be tedious and can lead to inefficient programs because threads are low-level, heavy constructs that are close to the hardware. Direct programming with threads forces you to do the work to efficiently map logical tasks onto threads. In contrast, the Threading Building
Blocks runtime library automatically schedules tasks onto threads in a way that makes efficient use of processor resources. The runtime is very effective at load-balancing the many tasks you will be specifying.

By avoiding programming in a raw native thread model, you can expect better portability, easier programming, more understandable source code, and better performance and scalability in general.

Indeed, the alternative of using raw threads directly would amount to programming in the *assembly language of parallel programming*. It may give you maximum flexibility, but with many costs.

**Threading Building Blocks targets** threading for performance

Most general-purpose threading packages support many different kinds of threading, such as threading for asynchronous events in graphical user interfaces. As a result, general-purpose packages tend to be low-level tools that provide a foundation, not a solution. Instead, Threading Building Blocks focuses on the particular goal of parallelizing computationally intensive work, delivering higher-level, simpler solutions.

**Threading Building Blocks is compatible with other threading packages**

Threading Building Blocks can coexist seamlessly with other threading packages. This is very important because it does not force you to pick among Threading Building Blocks, OpenMP, or raw threads for your entire program. You are free to add Threading Building Blocks to programs that have threading in them already. You can also add an OpenMP directive, for instance, somewhere else in your program that uses Threading Building Blocks. For a particular part of your program, you will use one method, but in a large program, it is reasonable to anticipate the convenience of mixing various techniques. It is fortunate that Threading Building Blocks supports this.

Using or creating libraries is a key reason for this flexibility, particularly because libraries are often supplied by others. For instance, Intel’s Math Kernel Library (MKL) and Integrated Performance Primitives (IPP) library are implemented internally using OpenMP. You can freely link a program using Threading Building Blocks with the Intel MKL or Intel IPP library.

**Threading Building Blocks emphasizes** scalable, data-parallel programming

Breaking a program into separate functional blocks and assigning a separate thread to each block is a solution that usually does not scale well because, typically, the number of functional blocks is fixed. In contrast, Threading Building Blocks emphasizes *data-parallel* programming, enabling multiple threads to work most efficiently together. Data-parallel programming scales well to larger numbers of processors by dividing a data set into smaller pieces. With data-parallel programming, program performance increases (scales) as you add processors. Threading Building Blocks also avoids classic bottlenecks, such as a global task queue that each processor must wait for and lock in order to get a new task.
Threading Building Blocks relies on generic programming

Traditional libraries specify interfaces in terms of specific types or base classes. Instead, Threading Building Blocks uses generic programming, which is defined in Chapter 12. The essence of generic programming is to write the best possible algorithms with the fewest constraints. The C++ Standard Template Library (STL) is a good example of generic programming in which the interfaces are specified by requirements on types. For example, C++ STL has a template function that sorts a sequence abstractly, defined in terms of iterators on the sequence.

Generic programming enables Threading Building Blocks to be flexible yet efficient. The generic interfaces enable you to customize components to your specific needs.

Comparison with Raw Threads and MPI

Programming using a raw thread interface, such as POSIX threads (pthreads) or Windows threads, has been an option that many programmers of shared memory parallelism have used. There are wrappers that increase portability, such as Boost Threads, which are a very portable raw threads interface. Supercomputer users, with their thousands of processors, do not generally have the luxury of shared memory, so they use message passing, most often through the popular Message Passing Interface (MPI) standard.

Raw threads and MPI expose the control of parallelism at its lowest level. They represent the assembly languages of parallelism. As such, they offer maximum flexibility, but at a high cost in terms of programmer effort, debugging time, and maintenance costs.

In order to program parallel machines, such as multi-core processors, we need the ability to express our parallelism without having to manage every detail. Issues such as optimal management of a thread pool, and proper distribution of tasks with load balancing and cache affinity in mind, should not be the focus of a programmer when working on expressing the parallelism in a program.

When using raw threads, programmers find basic coordination and data sharing to be difficult and tedious to write correctly and efficiently. Code often becomes very dependent on the particular threading facilities of an operating system. Raw thread-level programming is too low-level to be intuitive, and it seldom results in code designed for scalable performance. Nested parallelism expressed with raw threads creates a lot of complexities, which I will not go into here, other than to say that these complexities are handled for you with Threading Building Blocks.

Another advantage of tasks versus logical threads is that tasks are much lighter weight. On Linux systems, starting and terminating a task is about 18 times faster than starting and terminating a thread. On Windows systems, the ratio is more than 100-fold.
With threads and with MPI, you wind up mapping tasks onto processor cores explicitly. Using Threading Building Blocks to express parallelism with tasks allows developers to express more concurrency and finer-grained concurrency than would be possible with threads, leading to increased scalability.

**Comparison with OpenMP**

Along with Intel Threading Building Blocks, another promising abstraction for C++ programmers is OpenMP. The most successful parallel extension to date, OpenMP is a language extension consisting of pragmas, routines, and environment variables for Fortran and C programs. OpenMP helps users express a parallel program and helps the compiler generate a program reflecting the programmer’s wishes. These directives are important advances that address the limitations of the Fortran and C languages, which generally prevent a compiler from automatically detecting parallelism in code.

The OpenMP standard was first released in 1997. By 2006, virtually all compilers had some level of support for OpenMP. The maturity of implementations varies, but they are widespread enough to be viewed as a natural companion of Fortran and C languages, and they can be counted upon when programming on any platform.

When considering it for C programs, OpenMP has been referred to as “excellent for Fortran-style code written in C.” That is not an unreasonable description of OpenMP since it focuses on loop structures and C code. OpenMP offers nothing specific for C++. The loop structures are the same loop nests that were developed for vector supercomputers—an earlier generation of parallel processors that performed tremendous amounts of computational work in very tight nests of loops and were programmed largely in Fortran. Transforming those loop nests into parallel code could be very rewarding in terms of results.

A proposal for the 3.0 version of OpenMP includes tasking, which will liberate OpenMP from being solely focused on long, regular loop structures by adding support for irregular constructs such as while loops and recursive structures. Intel implemented tasking in its compilers in 2004 based on a proposal implemented by KAI in 1999 and published as “Flexible Control Structures in OpenMP” in 2000. Until these tasking extensions take root and are widely adopted, OpenMP remains reminiscent of Fortran programming with minimal support for C++.

OpenMP has the programmer choose among three scheduling approaches (static, guided, and dynamic) for scheduling loop iterations. Threading Building Blocks does not require the programmer to worry about scheduling policies. Threading Building Blocks does away with this in favor of a single, automatic, divide-and-conquer approach to scheduling. Implemented with work stealing (a technique for moving tasks from loaded processors to idle ones), it compares favorably to dynamic or guided scheduling, but without the problems of a centralized dealer. Static scheduling
is sometimes faster on systems undisturbed by other processes or concurrent sibling code. However, divide-and-conquer comes close enough and fits well with nested parallelism.

The generic programming embraced by Threading Building Blocks means that parallelism structures are not limited to built-in types. OpenMP allows reductions on only built-in types, whereas the Threading Building Blocks parallel_reduce works on any type.

Looking to address weaknesses in OpenMP, Threading Building Blocks is designed for C++, and thus to provide the simplest possible solutions for the types of programs written in C++. Hence, Threading Building Blocks is not limited to statically scoped loop nests. Far from it: Threading Building Blocks implements a subtle but critical recursive model of task-based parallelism and generic algorithms.

**Recursive Splitting, Task Stealing, and Algorithms**

A number of concepts are fundamental to making the parallelism model of Threading Building Blocks intuitive. Most fundamental is the reliance on breaking problems up recursively as required to get to the right level of parallel tasks. It turns out that this works much better than the more obvious static division of work. It also fits perfectly with the use of task stealing instead of a global task queue. This is a critical design decision that avoids using a global resource as important as a task queue, which would limit scalability.

As you wrestle with which algorithm structure to apply for your parallelism (for loop, while loop, pipeline, divide and conquer, etc.), you will find that you want to combine them. If you realize that a combination such as a parallel_for loop controlling a parallel set of pipelines is what you want to program, you will find that easy to implement. Not only that, the fundamental design choice of recursion and task stealing makes this work yield efficient scalable applications.

It is a pleasant surprise to new users to discover how acceptable it is to code parallelism, even inside a routine that is used concurrently itself. Because Threading Building Blocks was designed to encourage this type of nesting, it makes parallelism easy to use. In other systems, this would be the start of a headache.

With an understanding of why Threading Building Blocks matters, we are ready for the next chapter, which lays out what we need to do in general to formulate a parallel solution to a problem.