

Intel® Threading Building Blocks (Intel® TBB) 2.2

In-Depth

Contents

Intel® Threading Building Blocks (Intel® TBB) 2.2	3
Features	3
New in This Release	5
Technical Support	6
Which Intel TBB License is Right For Your Needs?.....	6

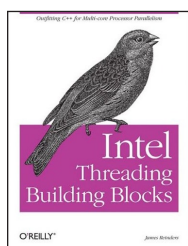
Intel® Threading Building Blocks (Intel® TBB) 2.2

Intel® Threading Building Blocks (Intel® TBB) is an award-winning C++ template library that abstracts threads to tasks to create reliable, portable, and scalable parallel applications. Use Intel TBB to implement task-based parallel applications and enhance developer productivity for scalable software on multicore platforms. Intel TBB is the most efficient way to implement parallel applications and unleash multicore platform performance compared with other threading methods like native threads and thread wrappers.

Productivity: Improves developer productivity by using task-based abstractions that make it easier to get scalable and reliable parallel applications with fewer lines of code (see figure 2). Task-based algorithms, containers, and synchronization primitives simplify parallel application development.

Future-proof applications: Application performance automatically improves as processor core count increases by using abstract tasks. Sophisticated task scheduler dynamically maps tasks to threads to balance the load among available cores, preserve locality, and maximize parallel performance.

Portability: Expand customer base by using a production-ready, open solution for parallelism that is available on a broad range of platforms. Available as a commercial and open source project, Intel TBB is coded in C++ and available on a multitude of platforms to provide a cross-platform solution for parallelism. Intel TBB is available as a standalone product, open source, or with the Intel® Compiler Professional Editions and Intel® Parallel Studio for a more complete and cost-effective solution.



Order the Intel® TBB Book at: http://www.amazon.com/Intel-Threading-Building-Blocks-Parallelism/dp/0596514808/ref=pd_bbs_sr_1?ie=UTF8&s=books&qid=1215536190&sr=1-1

Features

Intel TBB offers comprehensive, abstracted templates, containers, and classes for parallelism. Version 2.2 expands usage models and improves performance and usability. Figure 1 highlights the major functional groups within Intel TBB 2.2. Be sure to go to the What's New in 2.2 section on threadingbuildingblocks.org for a more detailed description of the new capabilities within Intel TBB 2.2.

Intel® Threading Building Blocks v2.2		
High-Level Threading Abstractions		
Parallel Algorithm Templates <ul style="list-style-type: none"> parallel_for parallel_for_each parallel_do parallel_while 	<ul style="list-style-type: none"> parallel_invoke parallel_reduce parallel_scan parallel_sort pipeline 	Thread-safe Concurrent Containers <ul style="list-style-type: none"> concurrent_vector concurrent_queue concurrent_bounded_queue concurrent_hash_map
Scalable Memory & Task Management		
Memory Allocation <ul style="list-style-type: none"> tbb_allocator cache_aligned_allocator zero_allocator aligned_space 	Task Scheduling <ul style="list-style-type: none"> task_scheduler_init task task_handle task_group structured_task_group tbb_thread 	Thread Local Storage <ul style="list-style-type: none"> combinable<T> enumerable_thread_specific
Low-Level Primitives		
Synchronization Primitives <ul style="list-style-type: none"> atomic<T> mutex queuing_mutex 	<ul style="list-style-type: none"> queuing_rw_mutex recursive_mutex spin_mutex 	Timing Primitive <ul style="list-style-type: none"> tick_count

Figure 1: Intel TBB offers comprehensive, abstracted templates, containers, and classes for parallelism. Version 2.2 expands usage models and improves performance and usability.

Intel TBB lets developers focus on adding value to their application instead of thread management. Figure 2 highlights a dramatically simpler multithreaded implementation with Intel TBB versus native threads. Intel TBB utilizes robust functions that reduce threading errors like deadlock and race conditions.

76% less code*! Focus on your app, not thread management

Windows* Threads	Intel® Threading Building Blocks
<p>Thread Setup and Initialization</p> <pre> CRITICAL_SECTION MyMutex, MyMutex2, MyMutex3; int get_num_cpus(void) { SYSTEM_INFO si; GetSystemInfo(&si); return (int)si.dwNumberOfProcessors; } int nthreads = get_num_cpus(); HANDLE *threads = (HANDLE *) alloca (nthreads * sizeof (HANDLE)); InitializeCriticalSection (&MyMutex); InitializeCriticalSection (&MyMutex2); InitializeCriticalSection (&MyMutex3); for (int i = 0; i < nthreads; i++) { DWORD id; @thread[i] = CreateThread (NULL, 0, parallel_thread, i, 0, 0); } for (int i = 0; i < nthreads; i++) { WaitForSingleObject (@thread[i], INFINITE); } </pre>	<p>2D Ray Tracing Application</p> <p>Thread Setup and Initialization</p> <pre> #include "tbb/task_scheduler_init.h" #include "tbb/spin_mutex.h" tbb::task_scheduler_init init; tbb::spin_mutex MyMutex, MyMutex2; </pre>
<p>Parallel Task Scheduling and Execution</p> <pre> const int MINPATCH = 150; const int DIVFACTOR = 2; typedef struct work_queue_entry_s { patch pch; struct work_queue_entry_s *next; } work_queue_entry_t; work_queue_entry_t *work_queue_head = NULL; work_queue_entry_t *work_queue_tail = NULL; void generate_work (patch *pch) { int startx, stopx, starty, stopy; int xx,yy; startx=pch->startx; stopx= pch->stopx; starty=pch->starty; stopy= pch->stopy; if((stopx-startx) >> MINPATCH) { int kpatchsize = (stopx-startx)/DIVFACTOR + 1; int ypatchsize = (stopy-starty)/DIVFACTOR + 1; for (yy=starty; yy<stopy; yy+=ypatchsize) for (xx=startx; xx<stopx; xx+=kpatchsize) { patch pch; pch.startx = xx; pch.starty = yy; pch.stopx = MIN(xx+kpatchsize-1,stopx); pch.stopy = MIN(yy+ypatchsize-1,stopy); generate_work (&pch); } } else { /* just trace this patch */ work_queue_entry_t *q = (work_queue_entry_t *) malloc (sizeof work_queue_entry_t); q->pch.startx = startx; q->pch.stopy = stopy; q->pch.starty = starty; q->pch.stopx = stopx; q->next = NULL; if (work_queue_head == NULL) work_queue_head = q; else { work_queue_tail->next = q; } work_queue_tail = q; } } void generate_worklist (void) { patch pch; pch.startx = startx; pch.stopx = stopx; pch.starty = starty; pch.stopy = stopy; generate_work (&pch); } bool schedule_thread_work (patch &pch) { EnterCriticalSection (&MyMutex3); work_queue_entry_t *q = work_queue_head; if (q != NULL) { pch = q->pch; work_queue_head = work_queue_head->next; } LeaveCriticalSection (&MyMutex3); return (q != NULL); } generate_worklist (); void parallel_thread (void *arg) { patch pch; while (schedule_thread_work (pch)) { for (int y = pch.starty; y <= pch.stopy; y++) for (int x=pch.startx; x<=pch.stopx; x++) { render_one_pixel (x, y); } if (scene.displaymode == RT_DISPLAY_ENABLED) { EnterCriticalSection (&MyMutex2); for (int y = r.rows().begin(); y != r.rows().end(); ++y) { GraphicsDrawRow(startx-1, y-1, totalx, (unsigned char *) &global_buffer[(y-starty)*totalx+(pch.startx-startx)*3]); } LeaveCriticalSection (&MyMutex2); } } } </pre>	<p>Parallel Task Scheduling and Execution</p> <pre> #include "tbb/parallel_for.h" #include "tbb/blocked_range2d.h" class parallel_task { public: void operator() (const tbb::blocked_range2d<int> &r) const { for (int y = r.rows().begin(); y != r.rows().end(); ++y) { for (int x = r.cols().begin(); x != r.cols().end(); x++) { render_one_pixel (x, y); } } if (scene.displaymode == RT_DISPLAY_ENABLED) { tbb::spin_mutex::scoped_lock lock (MyMutex2); for (int y = r.rows().begin(); y != r.rows().end(); ++y) { GraphicsDrawRow(startx-1, y-1, totalx, (unsigned char *) &global_buffer[(y-starty)*totalx+(y-starty)*3]); } } } }; parallel_for (tbb::blocked_range2d<int> (startx, stopx + 1, grain_size, startx, stopx + 1, grain_size), parallel_task ()); </pre> <p>This example includes software developed by John, E. Stone.</p> <p>* Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations.</p>

Figure 2: Side-by-side comparison of equivalent Windows* thread functionality that requires significantly more code to make a 2D ray tracing program, Tacheon, correctly threaded. Linux* and Mac OS* X developers can expect similar results.

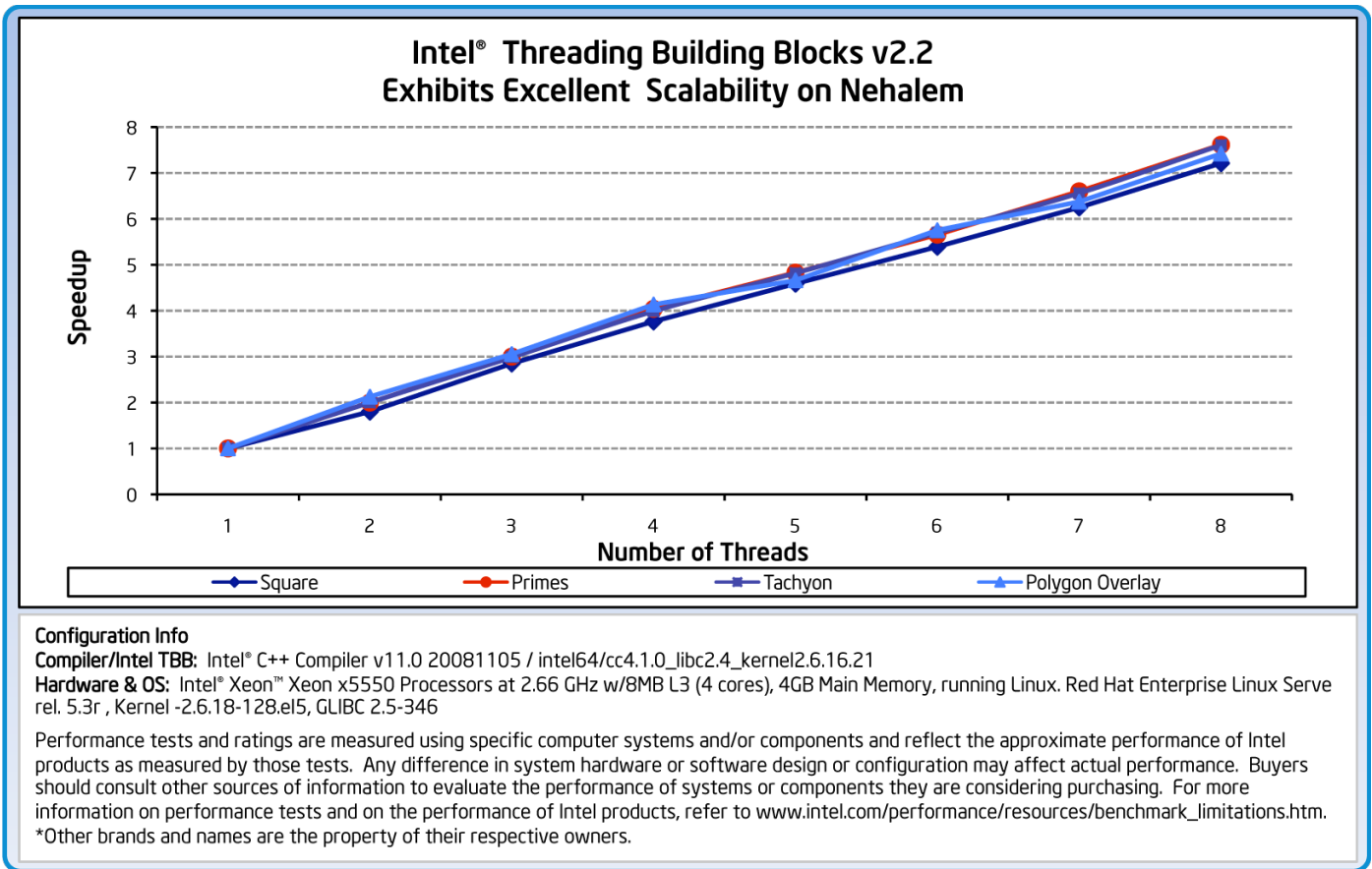


Figure 3: Excellent scalability and improved performance using Intel TBB versus a serial implementation. Linux and Mac OS X developers can expect similar results.

New in This Release

Intel TBB v2.2 offers several functionality, performance, and usability enhancements over 2.1.

Improved performance:

- Improved performance of scalable memory allocator
- Significant redesign of task scheduler for better performance and scalability
- Better performance of affinity partitioner
- `auto_partitioner` is the default for loop templates instead of `simple_partitioner`

New in scalable memory allocator:

- Intel® TBB 2.2 supports automatic replacement of OS allocator with its scalable memory allocator (Microsoft Windows* and Linux* OS)

Improvements in task scheduler:

- Intel® TBB 2.2 supports automatic initialization of task scheduler (`task_scheduler_init` is now optional)
- Support for task groups

New and improved parallel algorithms:

- New algorithms `parallel_invoke` and `parallel_for_each`
- New thread bound filter in pipeline
- Simplified interface for `parallel_for` for common loops
- Expanded support of Lambda expressions makes it easier to read and maintain code when using lambda capable compiler such as Intel® C++ Compiler 11.0 and above

New and improved data containers:

- New classes `enumerable_thread_specific` and `combinable` to support cross-platform thread local storage and algorithms for it
- Unbounded non-blocking interface for `concurrent_queue` and new blocking `concurrent_bounded_queue`
- Simplified interfaces for `concurrent_hash_map`
- Improved interfaces for `concurrent_vector`

Technical Support

Intel® Software Development Product purchases may include a year of support services, which provide access to Intel® Premier Support and all product updates during that time. Intel Premier Support gives you online access to technical notes, application notes, and documentation.

Which Intel TBB License is Right For Your Needs?

Intel TBB is available commercially as a binary distribution, and in open source in both source and binary forms. If you need commercial support services you should purchase either a standalone commercial license or take advantage of the considerable value in purchasing the Intel® Parallel Studio or Intel® Compiler Professional Edition. If your legal counsel is comfortable with your use of software under the Intel TBB open source license and you do not require commercial support services, please download the latest version of open source Intel TBB (<http://threadingbuildingblocks.org>). Finally, if you require the ability to modify or distribute the commercial source code of TBB, contact your Intel representative for more information.

When built from source, Intel TBB is intended to be highly portable, and so it supports a wide variety of operating systems and platforms. Binary distributions, including commercial distributions, are validated and officially supported for a variety of select hardware, software, operating systems, and compilers.

