Volume 11 Issue 04 Published, November 15, 2007 ISSN 1535-864X DOI: 10.1535/itj.1104.05

Multi-Core Software

The Foundations for Scalable Multi-Core Software in Intel® Threading Building Blocks

The Foundations for Scalable Multi-core Software in Intel® Threading Building Blocks

Alexey Kukanov, Performance, Analysis and Threading Lab, Intel Corporation
Michael J. Voss, Performance, Analysis and Threading Lab, Intel Corporation

Index words: threading building blocks, threading, scalability, parallelism, software

ABSTRACT

This paper describes two features of Intel® Threading Building Blocks (Intel® TBB) [1] that provide the foundation for its robust performance: a work-stealing task scheduler and a scalable memory allocator.

Work-stealing task schedulers efficiently balance load while maintaining the natural data locality found in many applications. The Intel TBB task scheduler is available to users directly through an API and is also used in the implementation of the algorithms included in the library.

In this paper, we provide an overview of the TBB task scheduler and discuss three manual optimizations that users can make to improve its performance: continuation passing, scheduler bypass, and task recycling. In the Experimental Results section of this paper, we provide performance results for several benchmarks that demonstrate the potential scalability of applications threaded with TBB, as well as the positive impact of these manual optimizations on the performance of fine-grain tasks.

The task scheduler is complemented by the Intel TBB scalable memory allocator. Memory allocation can often be a limiting bottleneck in parallel applications. Using the TBB scalable memory allocator eliminates this bottleneck and also improves cache behavior. We discuss details of the design and implementation of the TBB scalable allocator and evaluate its performance relative to several commercial and non-commercial allocators, showing that the TBB allocator is competitive with these other allocators.

INTRODUCTION

Performance-oriented developers now face the daunting task of threading their applications. Introducing parallelism into an application is a large investment. It is therefore imperative to implement a scalable solution, one that continues to increase performance, as the number of available cores and threads increases.

Intel TBB is a C++ template library that is designed to assist developers in porting their applications to multi-core platforms. The TBB library provides generic parallel algorithms [18] and concurrent containers [19] that enable users to write parallel programs without directly creating and managing threads. These algorithms are tested and tuned for the current generation of multi-core processors, and they are designed to scale as the core count continues to increase.

To provide efficient performance today and continued scalability tomorrow, the library is designed to support fine-grain parallelism through tasks. Tasks are user-level objects that are scheduled for execution by the TBB task scheduler. The task scheduler maintains a pool of native threads and a set of per-thread ready pools of tasks. At initialization, the TBB scheduler creates an appropriate number of threads in the pool (by default, 1 per hardware thread) and maintains the ready pools using a randomized work-stealing algorithm [2, 3].

In this paper, we describe the design of the TBB task scheduler and several scheduling optimizations users can keep in mind while coding their applications. In the Results section, we explore the scalability of TBB applications and highlight the impact of these scheduling optimizations on performance.

The task scheduler is complemented by the Intel TBB scalable memory allocator. In this paper, we provide an overview of its design and look at the tradeoffs. We compare its performance to several other commercial and non-commercial allocators.

RELATED WORK

The Intel TBB task scheduler is inspired by the early Cilk scheduler [2, 3]. Cilk is a parallel extension of the C programming language that defines additional keywords

and constructs. The Cilk project was a descendant of the Parallel Continuation Machine (PCM)/Threaded-C [13].

Both Cilk and the Intel TBB schedule lightweight tasks onto user threads. The Chare Kernel [14] is a portable set of functions that allows users to express parallelism in terms of small tasks (chares) with the runtime transparently managing resources. Unlike Intel TBB and Cilk, however, the Chare Kernel is targeted toward message passing machines.

Mainstream languages, such as those supported by the .NET CLR also recognize the need for thread pools, where users can submit tasks without the need to explicitly manage threads [15]. However, in the .NET CLR these thread pools are targeted at general-purpose applications and are not tuned for compute-intensive applications.

The McRT research program at Intel presented a software prototype of an integrated runtime library for large-scale chip-level multiprocessing (CMP) platforms [17], including a highly configurable, user-level scheduler. It can be used to realize a variety of co-operative scheduling strategies, including work stealing.

The design of the Intel TBB scalable allocator is based on contemporary research in scalable memory allocation [8, 9] and utilizes best-known design solutions; it has common roots with Hoard [8], LFMalloc, Vam [10], Streamflow [11] and other state-of-the-art concurrent and sequential allocators. The TBB scalable allocator is a productization of the scalable memory allocator developed as part of the McRT research program [7, 17].

THE TBB TASK SCHEDULER

The Intel TBB task scheduler is a *work-stealing scheduler*. The design of the TBB scheduler is inspired by the early Cilk scheduler, which Blumofe and Leiserson [2, 3] proved has optimal space, time, and communication bounds for well-structured ("fully strict") programs.

In a system that uses work-stealing, each thread maintains a local pool of tasks that are ready to run. Using local pools avoids the contention that may arise with the use of a global task queue. When executed, a task performs work and also may create additional tasks that are placed in the local pool. If a thread's pool becomes empty, it attempts to steal a task from another random thread's pool. This approach is in contrast to static scheduling methods where threads are assigned work up-front and from other dynamic scheduling methods where a central pool of tasks (or iterations) is maintained.

Blumofe and Leiserson [2, 3] showed that the expected parallel runtime of applications scheduled by the Cilk scheduler is $E[T_P] = O(T_1/P + T_\infty)$, where T_1 is the "work" or sequential time of the application, and T_∞ is the critical path length. This optimal bound shows that as P→ ∞, the expected time is only limited by the critical path length (the sequential part) of the application.

To achieve these same optimal bounds, the TBB task scheduler also uses a randomized work-stealing algorithm. An overview of its implementation is provided in the following section.

An Overview of the Task Scheduler Design

The TBB task scheduler evaluates *task graphs*. A task graph is a directed graph where nodes are tasks, and each node points to its *parent,* which is another task that is waiting on it to complete, or NULL. Each task has a *refcount* that counts the number of tasks that have it as their parent. Each task also has a *depth*, which is usually one more than the depth of its parent. The work of the task is performed by a user-defined function `execute` that is encapsulated within the task object.

To assist in providing an overview of the Intel TBB task scheduler, we use calculation of the nth Fibonacci number as a running example. A serial implementation of our Fibonacci example is shown below:

```
long SerialFib( long n ) {
 if( n<2 )
  return n;
 else
  return SerialFib(n-1)+SerialFib(n-2);
}
```

The function `ParallelFib`, shown below, uses the TBB task API to construct the root node of a task graph, an object of type `FibTask`. When this task's function `execute` is called, it will create two child tasks, also of type `FibTask`. Child `a` will calculate fibonacci(n-1) and child `b` will calculate fibonacci(n-2). When each of these tasks is executed, they will in turn recursively spawn child tasks as follows:

```
class FibTask: public task {
public:
 const long n;
 long* const sum;
 FibTask( long n_, long* sum_ ) :
     n(n_), sum(sum_) {}

 task* execute() {
  if( n<CutOff ) {
   *sum = SerialFib(n);
  } else {
   long x, y;
```

```
   FibTask& a = *new( allocate_child() )
       FibTask(n-1,&x);
   FibTask& b = *new( allocate_child() )
       FibTask(n-2,&y);
   set_ref_count(3);
   spawn( b );
   spawn_and_wait_for_all( a );
   *sum = x+y;
  }
  return NULL;
 }
};

long ParallelFib( long n ) {
 long sum;
 FibTask& a = *new(task::allocate_root())
     FibTask(n,&sum);
 task::spawn_root_and_wait(a);
 return sum;
}
```

For performance reasons, TBB requires users to set task refcounts explicitly with the `set_ref_count` call, instead of atomically incrementing it in `allocate_child`. The refcount should be set for a task before spawning any of its children.

Each task that spawns children waits at the `spawn_and_wait_for_all` call until all of its children complete. An additional guard reference is required for this, as shown in the above example by using the refcount of 3, while there are only 2 child tasks. A thread that enters a `spawn_and_wait_for_all` is free to execute other ready tasks while it waits.

In `ParallelFib`, after completing the wait call, the results of the child tasks are summed and returned. When `n<CutOff`, no additional child tasks are created and instead the leaf task will directly call SerialFib.

Figure 1 shows a snapshot of a task graph that might be created by an execution of ParallelFib. Tasks with non-zero reference counts (A, B, and C) must wait for their child tasks to complete before proceeding. The leaf tasks are ready to run.

As mentioned previously, the TBB library maintains a pool of threads, each of which has its own pool of ready tasks. Each per-thread task pool is implemented as an array of lists of tasks. A task goes into a pool only when it is deemed ready to run, i.e., it has been spawned and has a refcount of 0. Figure 2 shows a snapshot of a pool that corresponds to the task graph in Figure 1. Tasks A, B, and C do not appear in the pool because they have non-zero refcounts and therefore are not ready to run.

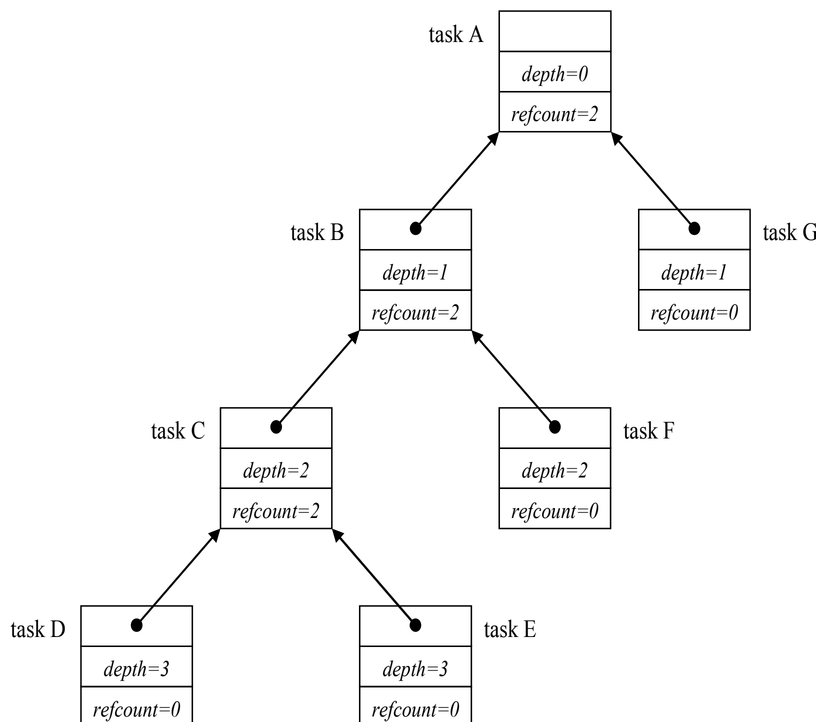

Figure 1: Intermediate task graph for the Fibonacci example

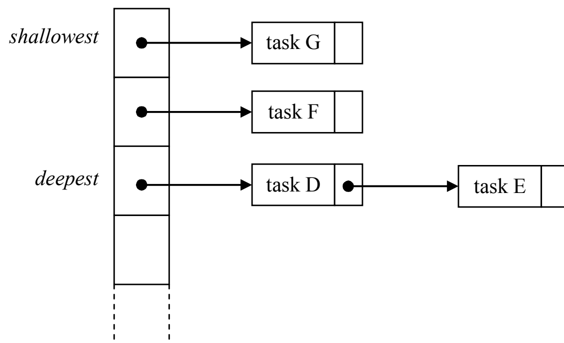

Figure 2: A pool of ready tasks that corresponds to the graph in Figure 1

Breadth-First Theft and Depth-First Work

The TBB task scheduler's fundamental strategy is "breadth-first theft and depth-first work." The breadth-first theft rule raises parallelism sufficiently to keep threads busy. The depth-first work rule keeps each thread operating efficiently once it has sufficient work to do.

A depth-first execution of a graph is the most efficient when performing a sequential execution because it provides better temporal locality and limits the space required for storing tasks. The deepest tasks are the most recently created tasks, and therefore are hottest in cache. When they complete, their parents can then execute, and although the parents are not hot in the cache, they're warmer than the tasks above them. A depth-first execution also limits the space required for storing tasks. When executing a node, only the nodes that lie along the path from the root to that node need to exist in memory.

Depth-first execution of a graph, however, limits parallelism. In contrast, always executing the shallowest tasks first leads to a breadth-first unfolding of the tree. This creates an exponential number of nodes that coexist simultaneously, providing ample tasks to steal but also excessively consuming memory.

To balance efficient execution and parallelism, the TBB scheduler therefore uses the "breadth-first theft and depth-first work" rule.

Each thread in the TBB thread pool executes a worker routine that actively looks for ready tasks to execute. A thread will first take the task at the front of the *deepest* list of its *own* pool[1]. If there are no ready tasks in its own pool and there is at least one non-empty task pool, it will then steal from the front of the *shallowest* list of *another* randomly chosen pool. If the chosen pool is empty, the thread tries to steal from another randomly selected thread until it succeeds.

[1] Optimizations will be discussed later that allow tasks to directly return a next task to execute, bypassing the task scheduler.

Scheduling Trade-offs and Optimizations

The Intel TBB task scheduler was inspired by the Cilk scheduler. Cilk is a parallel extension of the C programming language that defines additional keywords and constructs. Since Cilk requires a modified C compiler, it can rely on the compiler to perform Cilk-specific transformations and optimizations.

TBB on the other hand is a C++ template library and can be compiled using any standard-compliant C++ compiler. While this makes TBB more portable, it also means that correctness and performance cannot depend on any TBB-specific compiler passes. The TBB task API has therefore been designed to allow users to perform certain scheduling optimizations "manually" to achieve increased performance when necessary. The most important of these optimization opportunities are discussed below and their impact is evaluated in the Experimental Results section.

Minimizing Stack Use with Continuation Tasks

As mentioned before, TBB uses a "breadth-first theft and depth-first work" approach. However, this approach can sometimes cause the processor stack to overflow.

For example, consider the case when a task enters a `spawn_and_wait_for_all`. The task cannot continue until all of its children complete. On entering the wait, the calling thread is released to execute or steal other tasks. If it steals the shallowest task from another thread, it then begins a depth-first execution of this stolen tree.

However, the initial task that entered the `spawn_and_wait_for_all` is kept on the processor stack to maintain its local storage and instruction pointer. The newly stolen tree then begins to unfold on top of the waiting subtree on the processor stack. This situation could occur repeatedly, causing the stack to overflow.

To avoid this situation, the TBB task scheduler forces a thread to only steal tasks that are deeper than any waiting task. While this limits stack growth, it also limits the choice of tasks to steal and therefore might limit parallelism.

To avoid restricting the choice of tasks to steal while at the same time limiting stack space growth, the TBB task interface allows developers to specify *continuation tasks*. A task can replace itself in the graph with a continuation task and then return, freeing up its stack space. When the children complete, the continuation task is spawned to finish the work delegated to it by the parent.

To use a continuation task `c`, the children of a task are allocated as children of `c` and not the task itself. Like

other tasks, `c` becomes ready to run when its children complete and will only then be spawned. The code for spawning children using "continuation-passing" for our Fibonacci example is shown below:

```
FibContinuation& c =
   *new( allocate_continuation() )
    FibContinuation(sum);
FibTask& a = *new( c.allocate_child() )
    FibTask(n-2,&c.x);
FibTask& b = *new( c.allocate_child() )
    FibTask(n-1,&c.y);
c.set_ref_count(2);
c.spawn( b );
c.spawn( a );
return NULL;
```

The implementation of `class FibContinuation` (not shown) inherits from `class task`, and sums `c.x` and `c.y` into `sum` in its `execute` function. The benefit of this approach is that after spawning children tasks in FibTask, the `execute` function returns, removing itself from the stack. Only the tasks that are actively executing are on the processor stack.

While there are benefits to the use of continuation tasks, there are also downsides. When using continuation tasks, all live state passed from parent to child cannot reside in the parent or its `execute` stack frame, since the parent may be destroyed before the child completes. Therefore additional care may be needed to properly encapsulate the live state of the computation. Also continuation passing requires the creation of an additional task object. In fine-grain tasks, this additional runtime overhead of task creation might be noticeable.

Reducing Overheads: Scheduler Bypass and Task Recycling

Luckily once an algorithm is using continuation tasks, it can also make use of two other overhead reducing techniques: *scheduler bypass* and *task recycling*.

With scheduler bypass, a task's `execute` function explicitly returns the next task to execute. Since the next task is known, the more complex logic to select a task is avoided in the scheduler's code. To use scheduler bypass in our Fibonacci example, the child task `a` is not spawned but is instead returned as shown below:

```
c.spawn( a );
return &a;
```

Once continuation passing and scheduler bypass are in use, it also becomes possible to recycle task objects. Normally when a task returns from its `execute` function, the task object is automatically deallocated. However, a user can choose to recycle a task object, making it live beyond the return and avoiding the repeated allocation and deallocation of task objects. Recycling a task as one of its own children is shown below for Fibonacci:

```
FibContinuation& c =
   *new( allocate_continuation() )
    FibContinuation(sum);
FibTask& b = *new( c.allocate_child() )
    FibTask(n-1,&c.y);
recycle_as_child_of(c);
n -= 2;
sum = &c.x;
c.set_ref_count(2);
c.spawn( b );
return this;
```

As shown in the Experimental Results section, scheduler bypass and task recycling often more than make up for the extra overhead added from the allocation of a continuation task.

SCALABLE MEMORY ALLOCATION

Until recently, mainstream client applications have targeted single-processor PCs. Therefore state-of-the-art general-purpose memory allocators such as Doug Lea's dlmalloc [4] have evolved to optimize for the sequential case. They were designed with two main principles in mind: efficient use of memory space and minimization of CPU overhead. Unfortunately, design decisions made to achieve these principles often hinder these allocators from providing good parallel performance.

Even the best sequential allocator can easily become a performance bottleneck in a parallel application. To ensure correctness, access to its heap must be properly protected. Using a single global lock for protection would amount to serializing all allocations. Detlefs et al. [5] showed that real applications spend up to 20% of execution time in memory allocator routines (even more with inefficient allocators). According to Amdahl's law [6], an application that is 20% sequential can never achieve more than a 5x speedup, even when using an infinite number of cores. Serializing allocations is therefore clearly not a scalable solution. Though more advanced schemas were developed to adapt dlmalloc for multi-threaded applications [12, 16], their scalability is also limited [8, 9, 12].

In addition, while space and CPU efficiency remain considerations in the design of a scalable memory allocator, they are not as important as before. The larger memory sizes available in the average PC and the growing speed gap between CPU and memory bring other

considerations to the forefront, such as cache locality and prevention of false sharing.

Unfortunately, malloc implementations supplied by widely used C runtime libraries such as glibc and the Microsoft Visual C++* RTL still do not provide proper scalability for multi-threaded applications. As Intel TBB aims to ease the development of efficient and scalable parallel applications, it is unable to rely on these by-default allocators, and therefore provides its own scalable memory allocation library.

The TBB Scalable Allocator

The TBB scalable allocator is a productization of the scalable memory allocator developed as part of the McRT research program at Intel [7, 17].

In TBB, we improved the McRT code for better portability (for example, we had to rework the parts depending on other components of the McRT library) and addressed the performance of some corner case situations that were ignored by the research project. However, the major structure of the TBB scalable allocator is the same as the McRT design.

Figure 3 shows the high-level design of the TBB scalable allocator.

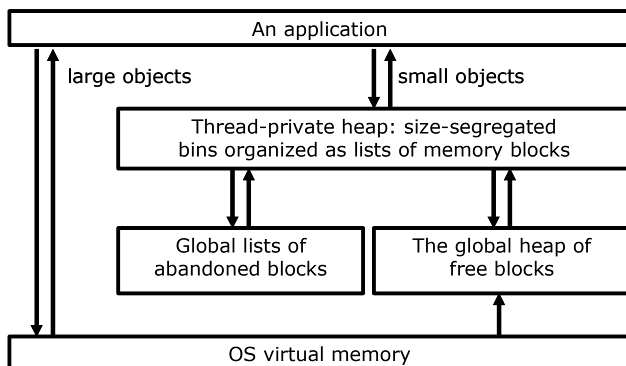

Figure 3: High-level design of the scalable allocator

The allocator requests memory from the OS in 1MB chunks and divides each chunk into 16K-byte aligned *blocks*[2]. These blocks are initially placed in the global heap of free blocks. Currently, requested memory is never returned to OS (except for large allocations as described below), so the allocator carefully ensures that memory is reused. New blocks are only requested when a thread can't find any free *objects* in the blocks of its own heap and there are no available blocks in the global heap.

As in some other allocators, requests for large objects are redirected straight to OS virtual memory services. In the TBB allocator, the border between large and "regular" sizes lies slightly below 8K. However, we found that for better competitiveness, memory pieces of 8K to ~64K size should also be cached; explicitly managing these sizes is part of the future work for TBB.

Like many other widely used concurrent allocators, the TBB allocator uses *thread-private heaps*. Such a design has proven to cut down on the amount of code that requires synchronization, and reduce false sharing, thus providing better scalability. Each thread allocates its own copy of heap structures and accesses it via thread-specific data (TSD) using corresponding system APIs.

The heaps are *segregated*, i.e., they use different storage bins to allocate objects of different sizes. A memory request size is rounded up to the nearest object size. This technique provides better locality for similarly-sized objects that are often used together (for example, imagine an application traversing over a list or a tree). In the TBB allocator, the difference between consecutive object sizes in general does not exceed 25%, so internal fragmentation remains reasonable.

Figure 4 illustrates the internal design of a bin. A bin only holds objects of a particular size, and it is organized as a double-linked list of blocks. At each moment, there is at most one *active block* used to fulfill allocation requests for a given object size. Once the active block has no more free objects, the bin is switched to use another block.

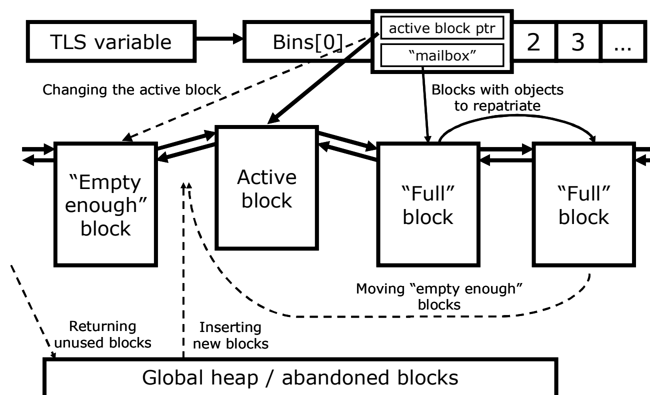

Figure 4: Design of a storage bin in a thread-private heap

Unlike in other allocators, the active block may be located in the middle of the list; *empty enough*[3] blocks are placed before it, and *full* blocks are placed after it. This design

[2] Following the authors of the McRT malloc [7], we will use terms "object" and "block"; in other literature, they can be called "block" and "superblock," respectively.

[3] A block counts as "empty enough" if the share of allocated objects drops below the predefined threshold.

minimizes the time to search for a new block if the active one is full. When enough objects are freed in a block, the block is moved right before the active block and thus becomes available for further allocation. A block with all its objects freed returns back to the global heap; new blocks are taken from there as required.

The design decisions made at higher levels allow certain optimization techniques for object allocation. With thread-local heaps, the common allocation path does not contain synchronization apart from the TSD access managed by the OS; the same is true for deallocation of a thread's own objects, as shown below.

Size segregation and aligned blocks have made per-object headers needless; all information required to free an object can be easily obtained via the *block header*. As a result, objects are tightly packed in the block (as shown in Figure 5), which leads to a potentially smaller memory footprint and better cache locality.

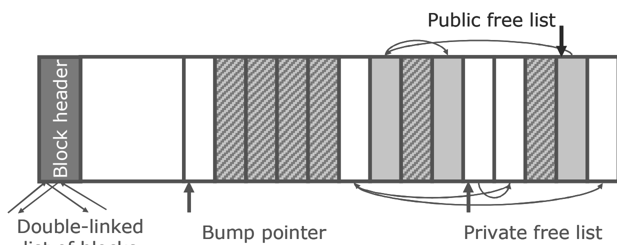

Figure 5: Structure of a memory block containing allocation objects of a specific size

Berger et al. [8] proved that allocators with pure private heaps cause unbounded memory blowup in producer-consumer applications. To avoid this, memory should be returned to the heap it was allocated from. In the TBB scalable allocator, an object is naturally returned to its enclosing block. However doing so means that a *foreign thread*[4] can interfere with operations of the owning thread, possibly leading to slowdown. To avoid that, two separate free lists are used for objects returned by the owner and by foreign threads.

Allocation requests are usually served from the *private free list* and so do not require synchronization; only when the request cannot be satisfied this way are the public free lists inspected. Unlike in McRT malloc [7], we do not make repatriation of objects completely non-blocking due to portability restrictions and stricter requirements; we use fine-grained locks that are distributed as much as possible.

[4] A thread returning a memory object to the block owned by another thread.

EXPERIMENTAL RESULTS

In this section, we present performance data to evaluate the performance of both the Intel TBB task scheduler and the scalable memory allocator. All results were collected on a server system with two Quad-Core Intel® Xeon® processors X5355[5] running Red Hat Enterprise Linux 4 (update 4). We present data using 1 through 8 threads to show performance on both a small number of cores as well as to show the scalability beyond the number of cores available in a single multi-core processor today.

Performance of the Task Scheduler

In this section, we present the scalability of several benchmarks, highlighting the impact of continuation passing, scheduler bypass, and task recycling on the performance of each application.

Methodology

To evaluate the performance of the TBB scheduler as well as the impact of the manual optimization described above, we show results for applications using TBB without scheduling optimization (TBB); using only continuation passing (TBB+C); using continuation passing and scheduler bypass (TBB+CB); and using continuation passing, scheduler bypass, and task recycling (TBB+CBR). For each benchmark we show the speedups relative to an optimized serial implementation that does not use TBB.

Benchmark Descriptions

We use four applications to evaluate the performance of the task scheduler:

- **fibonacci.** The Fibonacci benchmark corresponds to the running example provided above in the description of the task scheduler. In our benchmark runs, we calculate the 50th Fibonacci number, with serial cutoffs of 12 and 20.
- **parallel_for.** This microbenchmark uses an Intel TBB parallel_for algorithm to iterate over a range of 100 million integers applying an empty loop body to each element. In the TBB library, all three scheduling optimizations are used by default. To allow the performance impact of the various optimizations to be

[5] Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

measured, the implementation of parallel_for was modified to allow the selective disabling of specific optimizations.

- **sub_string_finder.** This benchmark is an example that is provided with the TBB library. The application calculates, for each position in a string, the location of the largest substring found elsewhere in the string that matches a string starting at the current position. The code uses the modified parallel_for described above to isolate the impact of the scheduling optimizations.
- **tacheon.** Tacheon is a 3D ray tracer that is distributed as another example with the TBB library. The code also uses the parallel_for algorithm modified to allow selective disabling of optimizations.

Benchmark Results

Figure 6 shows the performance of the Fibonacci example when executed on 1 through 8 threads on the aforementioned server. In the tests we used serial cutoffs of 12 and 20. When calculating the 50th Fibonacci number, the overhead of task creation is small when using a cutoff of 20, as shown by the speedup of 1 when using 1 thread. The scalability for this case is also excellent, with a speedup of nearly 8 on 8 threads.

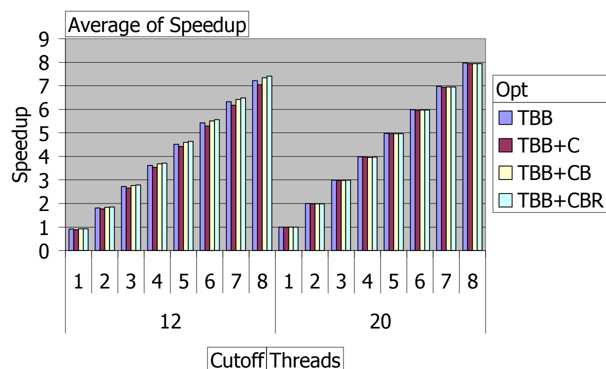

Figure 6: The speedup of the Fibonacci example when using different scheduling optimizations and serial cutoff values. The performance on 1 through 8 threads is reported for each configuration.

With a cutoff of 12, however, finer-grain tasks are created resulting in a noticeable scheduling overhead and a speedup of only 0.93 on 1 thread. With measurable overheads, the impact of the scheduling optimizations can also be seen. As discussed above, the use of continuation passing may provide additional opportunities for stealing but requires the allocation of additional task objects, often resulting in a slowdown. This effect is clearly seen in the TBB+C bars in Figure 6. However continuation passing also enables the scheduler bypass and task recycling optimizations, which when combined, result in speedups beyond the simple TBB case. On 8 threads, the speedup increases from 7.2 with no optimizations to 7.4 with all optimizations, an increase of approximately 3%.

The performance of the parallel_for microbenchmark is shown in Figure 7. The parallel_for algorithm creates tasks that apply a user-provided body to subranges of the user-provided range. When using the parallel_for algorithm, developers may explicitly specify a grainsize or choose to use the *auto_partitioner*. If a grainsize is specified, the default parallel_for algorithm recursively divides the provide range until the subranges are less than the grainsize. Tasks are created that apply the body to these subranges. If the auto_partitioner is used, the library adaptively tries to select a good partitioning of the range.

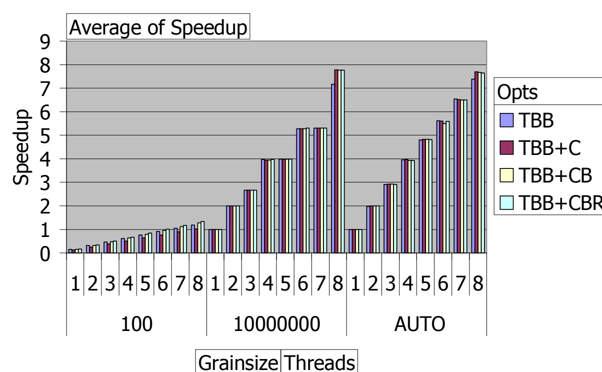

Figure 7: The speedup of the parallel_for benchmark when using different scheduling optimizations and grainsizes. The AUTO configurations use the auto_partitioner to divide the loop iterations; the other configurations use the provided grainsize parameter with the simple_paritioner. The performance is reported on 1 through 8 threads.

In Figure 7, results for a grainsize of 100, a grainsize of 10,000,000, and the auto_partitioner are shown. Again, for a large grainsize (and the correspondingly large-grain tasks) the overhead of the scheduler is negligible and the speedup on 8 threads is close to 8. Interestingly, the lack of available parallelism limits speedup even for large tasks, as demonstrated by the speedup increase with continuation passing over the base unoptimized case.

The fine-grain tasks, of only 100 iterations of an empty loop body, show high overhead (a speedup of 0.15 on 1 thread and 1.19 on 8 threads). Again because of the visibility of overheads, the impact of scheduler bypass and task recycling is clear on 1 through 8 threads. The speedup of 1.19 on 8 threads is improved to 1.34 when all three optimizations are applied.

Figures 8 and 9 present the performance of two larger, more realistic benchmarks. In both of these benchmarks, the performance using the default grainsize of 100 for

sub_string_finder and of 50 for tacheon is measured as well as a grainsize of 1. In both applications, the scheduling overhead shown in the 1-thread case is small even when a grainsize of 1 is used. The scalability of both applications is also good, with a speedup of close to 8 for sub_string_finder and a speedup of 7.7 for tacheon.

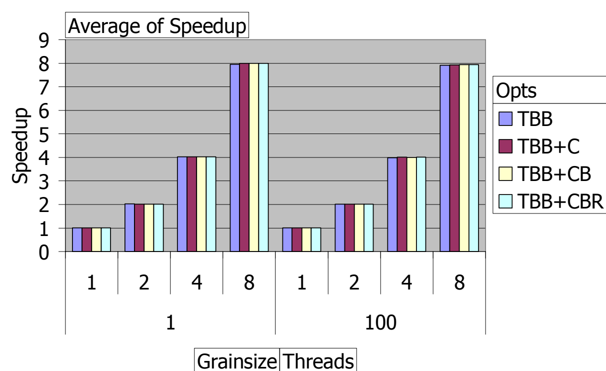

Figure 8: The speedup of the sub_string_finder example using different scheduling optimizations and grainsize parameters. The performance on 1, 2, 4, and 8 threads is presented.

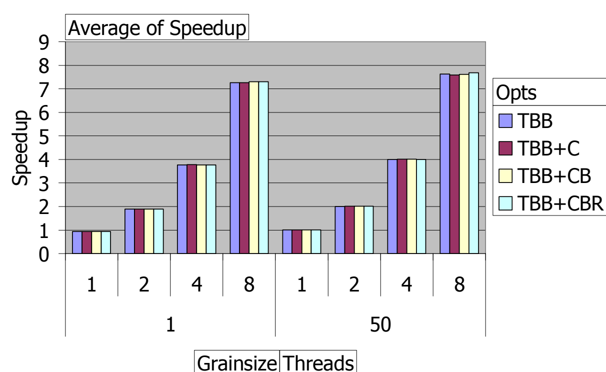

Figure 9: The speedup of the tacheon example using different scheduling optimizations and grainsize parameters. The performance on 1, 2, 4, and 8 threads is presented.

In summary, the scalability of the TBB scheduler is shown to allow linear speedups for several small benchmarks.[6] It is also clear that the overhead of the TBB scheduler is seen for fine-grain tasks (for example 100 iterations of an empty loop). When these overheads are visible, continuation passing alone often leads to a slowdown relative to the unoptimized case. However, continuation passing can be applied to enable scheduler bypass and task recycling, which are consistently shown to improve performance when scheduling fine-grain tasks.

[6] The speedup of other applications will vary depending on application characteristics.

Performance of Memory Allocation

In this section, we present the comparative performance data for the TBB scalable allocator and five other commercial and non-commercial memory allocators.

Memory Allocators being Compared

The TBB scalable memory allocator binaries were obtained from tbb20_010oss_lin.tar.gz package available at http://www.threadingbuildingblocks.org.

Other allocators in the comparison are these:

- The default memory allocator of GNU C runtime library (glibc) v2.3.4.
- Google's TCMalloc (google-perftools v0.92) from http://code.google.com/p/google-perftools built by gcc 3.4.6.
- Hoard v3.6.2 taken from http://www.hoard.org, also built by gcc 3.4.6.
- Memory Tuning System* (MTS) binaries provided by NewCode Technologies, Inc., http://www.newcodeinc.com.
- SmartHeap* for SMP binaries provided by MicroQuill, http://www.microquill.com.

Benchmark Description

When comparing memory allocators, it makes sense to use different tests that exercise different aspects of memory allocation routines. We used four benchmarks in our study: the Larson benchmark, the MTS demo test, and two internally developed microbenchmarks, speed-cross and false-sharing.

The **false-sharing micro-benchmark** was developed to check for the performance penalty due to false sharing induced by an allocator. Each thread repeatedly allocates a small object of a given size, then writes and reads every byte in the object many times in a loop and measures the time of the loop. The result is reported for every thread. If objects allocated by different threads share the same cache line, there should be a significant time penalty.

The **speed-cross micro-benchmark** was developed as a stress-test of the multi-threaded behavior of an allocator. Each thread repeatedly allocates a chunk of memory objects, touches each one by reading and writing a few bytes, and then transmits these objects in equal proportion to all other threads. Then each thread deallocates the objects it just received. Thus all objects are freed by foreign threads, and all subsequent allocations potentially

reuse these objects. The test reports average allocation and deallocation time per 1,000 objects.

Unlike the microbenchmarks intended to check specific aspects of memory allocation, the other two tests try to exercise memory in a more or less realistic way.

The **MTS demo test** was obtained from the MTS evaluation package. It attempts to mimic typical allocation behavior of applications by requesting few large objects, more medium-size objects, and significantly more small objects. The test measures elapsed time in seconds.

The **Larson benchmark** was originally developed by Larson and Krishnan [12] to model the allocation behavior of a multi-threaded server and test its throughput as the number of malloc and free pair operations per second. We took the benchmark from http://www.hoard.org.

Benchmark Results

The internal micro-benchmark data presented below were collected for objects of the machine word size, i.e., eight bytes on our test server.

The false-sharing benchmark demonstrates that of all the tested allocators, only the glibc allocator induces false sharing. Figure 10 shows execution time for various numbers of running threads as the percent of difference from the single-threaded run. While the test slowed down by 40-50% when executed with the default allocator, the difference is within 10% for all of the other allocators.

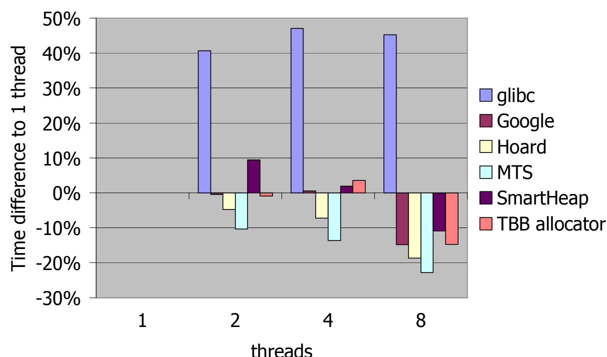

Figure 10: The difference in execution time of the false-sharing benchmark, running on 2, 4, and 8 threads, to the time of the single-threaded run, for various memory allocators

In addition, the test was faster with multiple threads which is especially well observed for eight threads. This effect could be possibly explained by decreased thread migration between cores when the number of active threads increases.

The speed-cross benchmark heavily stresses the allocators by freeing every object in a thread other than the one it was allocated; it's truly a worst-case test. In Figure 11, the summary time[7] of malloc and cross-thread free operations is shown for various numbers of threads. For allocators returning memory pieces to the heap of the allocating thread, the internal contention increases with the growing number of threads. The chart demonstrates that the TBB scalable allocator keeps being faster than the others with a growing number of threads and increasing contention.

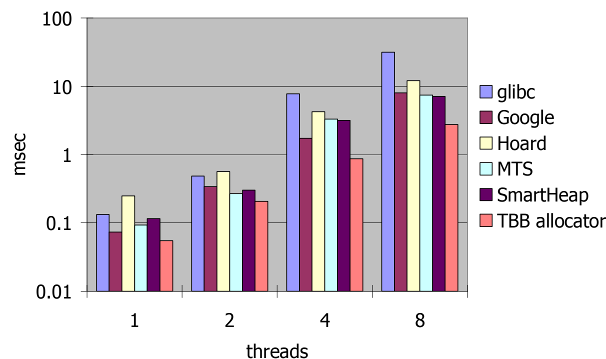

Figure 11: The average time to allocate and free 1,000 objects in the speed-cross benchmark is presented for 1, 2, 4, and 8 threads. The chart uses logarithmic scale.

Figure 12 demonstrates the elapsed time to run as reported by the MTS demo test.

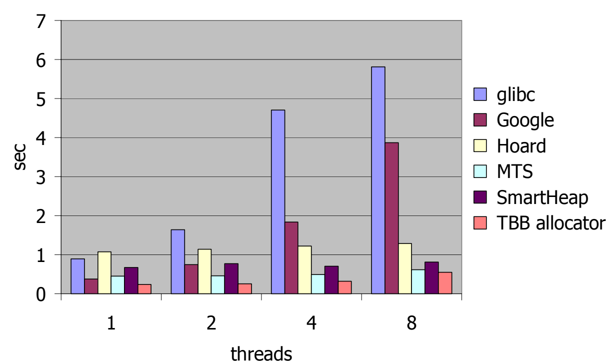

Figure 12: The elapsed time of the MTS demo test running on 1, 2, 4, and 8 threads, for various memory allocators

It is clearly seen that the test slows down as the number of threads increases for both the glibc malloc and Google's allocator; obviously their performance does not scale in this test. Other allocators scale well enough, though the test performance drops faster with the TBB allocator than with Hoard and the two commercial allocators. We are

[7] Due to nature of the benchmark, it separately collects data for malloc and free, then sums them up.

currently investigating the source of this performance drop.

The Larson benchmark results are shown in Figure 13. The benchmark parameters were set to allocate small size objects of 8 to 100 bytes. The TBB allocator scales linearly in this test with the best speedup slope. With 8 threads, it provides a 6x increase in throughput. Also note that the glibc malloc experienced a drop in throughput with multiple threads running. As in the other tests before, the Larson benchmark gives additional proof that the default glibc allocator can be a bottleneck in parallel code.

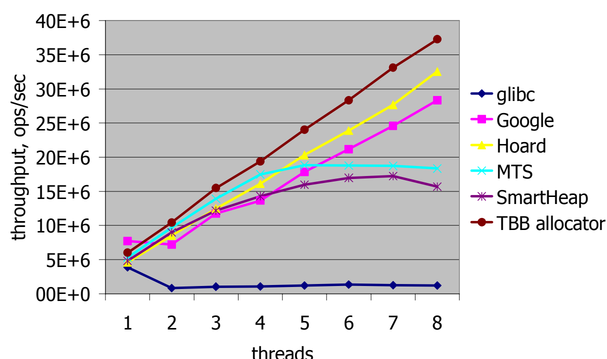

Figure 13: The throughput, in allocations per second, of the Larson benchmark running on 1 through 8 threads for various memory allocators

To summarize, all examined allocators except the glibc malloc showed their eligibility for parallel applications, and there is no single winner. While not always the best, the TBB scalable allocator performed competitively in all our tests.

Combined Performance of the Task Scheduler and the Scalable Allocator

In this section, we show the impact of the scalable allocator combined with an analysis of the impact of the task-scheduling optimizations. For this analysis, we use the tree sum example application provided in the TBB library distribution.

The tree sum application first generates a binary tree that contains nodes each holding a float value. It then performs a summation of the values in the tree. Both phases are done in parallel using TBB tasks, with a serial cutoff value below which the subtrees are allocated or summed sequentially.

Figure 14 shows the performance of the tree allocation phase of the benchmark when using both the scalable allocator and the default malloc implementation. Results are provided for a serial cutoff of both 100 nodes and 10,000 nodes.

First, it is clear that the allocation phase scales as additional threads are used only when the scalable allocator is employed. The performance of the standard malloc version degrades as additional threads are used.

Second, the impact of the scheduling optimizations is again demonstrated by the finer grained tasks (a cutoff of 10 nodes). There is an initial loss for employing continuation passing, but this loss is mitigated by the additional application of scheduler bypass and task recycling. And as expected, the larger tasks of 10,000 nodes show negligible impact from the optimizations.

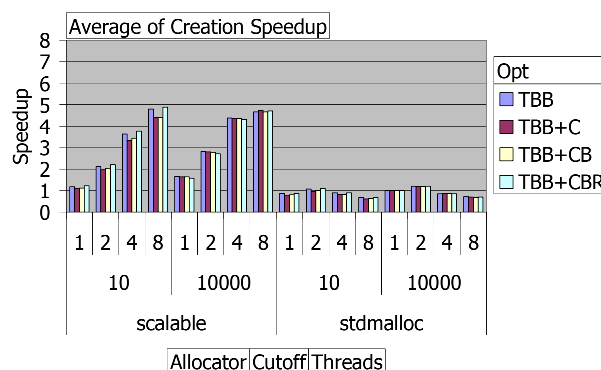

Figure 14: The speedup of the tree allocation phase of the tree_sum example using the scalable allocator and the default malloc implementation. The impact of the various scheduling optimizations is also shown. The performance on 1, 2, 4, and 8 threads is shown when using a serial cutoff of 10 and 10,000.

Figure 15 shows the performance of the summation phase of tree sum. Because of the locality and false-sharing benefits of the scalable allocator, the performance and scalability of the computation are also better than with the standard malloc implementation. The impact of the manual scheduling optimizations is also seen here for fine-grain tasks, and it is negligible for large-grain tasks.

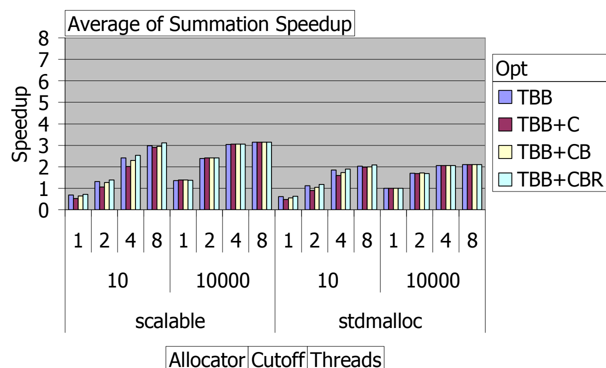

Figure 15: The speedup of the tree summation phase of the tree_sum example using the scalable allocator and the default malloc implementation. The impact of the various scheduling optimizations is also shown. The performance on 1, 2, 4, and 8 threads is shown when using a serial cutoff of 10 and 10,000.

CONCLUSION

Intel Threading Building Blocks is a C++ template library designed to raise the level of abstraction for parallelism as developers port their code to multi-core platforms. Starting with the 2.0 version, Intel TBB is also provided at www.threadingbuildingblocks.org as an open-source project licensed under the GNU Public License.

Two key features of the library are its work-stealing task scheduler and scalable memory allocator. Both of these systems reduce the need of users to understand the many complex issues related to multi-core performance and scalability.

In the TBB Task Scheduler section, we provided an overview of the task scheduler design and outlined several manual optimizations that users can perform to improve the performance of the scheduler when executing fine-grain tasks.

In the Scalable Memory Allocation section, we described the motivation for and implementation of the scalable memory allocator, highlighting the design characteristics that decrease synchronization, increase locality, and avoid false sharing.

In the Experimental Results section, we explored the performance of a number of benchmarks on a server with two Quad-Core Intel Xeon processors. We showed that the overhead of work stealing is low for large-grain tasks, and that the manual optimizations described in this paper offer a small but noticeable improvement when scheduling fine-grain tasks.

In our evaluation of scalable memory allocators, the TBB scalable allocator was shown to be competitive with several commercial and research allocators.

In an analysis of an example that studied the combined effects of the scheduling optimizations and the scalable allocator, the use of the scalable allocator showed a large impact for both small- and large-grain tasks. The scheduling optimizations were shown to have a small performance impact for the small-grain tasks and a negligible impact on the scheduling of the larger-grain tasks. This confirms the assertion that memory allocation can sometimes be a limiting factor in the scalability of parallel applications and that a scalable allocator can remove this bottleneck.

With the growing availability of multi-core platforms, it is becoming imperative for performance-oriented developers to thread their code. Intel TBB, built on its work-stealing task scheduler and scalable memory allocator, offers an exciting solution to ease the burden of this transition.

ACKNOWLEDGMENTS

We first must acknowledge the architect of Intel Threading Building Blocks, Arch Robison, who did much of the background research and design of the task scheduler. We also acknowledge the other members of the TBB development team: Elena Gavrina, Chris Huson, Anton Malakhov, Andrey Marochko, Alexey Murashov, Anton Pegushin, Vladimir Polin, and Dave Poulsen. As mentioned in the paper, the TBB scalable allocator is a productization of the McRT allocator, so we acknowledge the contributions of Richard Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. We thank NewCode Technologies, Inc. and MicroQuill for providing evaluation copies of their memory allocators. We also thank the reviewers of this paper for their many useful and insightful comments.

REFERENCES

[1] James Reinders, *Intel Threading Building Blocks*, O'Reilly Media, Inc, Sebastopol, CA, 2007.

[2] Robert D. Blumofe and Charles E. Leiserson, "Scheduling Multithreaded Computations by Work-Stealing," in *Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science,* Sante Fe, New Mexico, November 20–22, 1994.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall and Yuli Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '95), Santa Barbara, California, July 19–21, 1995.

[4] Doug Lea, "A Memory Allocator," at http://gee.cs.oswego.edu/dl/html/malloc.html

[5] David Detlefs, Al Dosser, and Benjamin Zorn, "Memory Allocation Costs in Large C and C++ Programs," *Software Practice and Experience,* 24(6), pp. 527–542, June 1994.

[6] Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," *AFIPS Conference Proceedings,* (30), pp. 483–485, 1967.

[7] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg, "McRT-Malloc – A Scalable Transactional Memory Allocator," in *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, pp. 74–83, Ottawa, Canada, June 2006.

[8] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 117–128, November 2000.

[9] Maged M. Michael, "Scalable Lock-free Dynamic Memory Allocation," in *Proceedings of the ACMSIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 35–46, Washington, D.C., June 2004.

[10] Yi Feng and Emery D. Berger, "A Locality-Improving Dynamic Memory Allocator," in *Proceedings of the Third Annual ACM SIGPLAN Workshop on Memory Systems Performance*, pp. 68–77, Chicago, IL, June 2005.

[11] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos, "Scalable Locality-Conscious Multithreaded Memory Allocation," in *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, pp. 84–94, Ottawa, Canada, June 2006.

[12] Paul Larson and Murali Krishnan, "Memory Allocation for Long-Running Server Applications," in *Proceedings of the First International Symposium on Memory Management,* pp. 176–185, Vancouver, BC, October 1998.

[13] Michael Halbherr, Yuli Zhou and Christopher F. Joerg, "MIMD-style parallel programming with continuation-passing threads," in *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications,* Capri, Italy, September 1994.

[14] W. Shu and L. V. Kale, "Chare Kernel – A Runtime Support System for Parallel Computations," *Journal of Parallel and Distributed Computing,* 11(3), Academic Press, pp. 198–211, 1991.

[15] Jeffrey Richter, ".NET: The CLRs Thread Pool," *msdn Magazine,* 18(6), June 2003.

[16] Wolfram Gloger, "Dynamic Memory Allocator Implementations in Linux System Libraries," at http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html

[17] Bratin Saha et al., "Enabling scalability and performance in a large scale CMP environment," in *Proceedings of the 2007 conference on EuroSys*, pp. 73–86, Lisbon, Portugal, March 2007.

[18] Michael Voss, "Demystify Scalable Parallelism with Intel Threading Building Block's Generic Parallel Algorithms," DevX.com, Jupiter Media, October 2006, at http://www.devx.com/cplus/Article/32935.

[19] Michael Voss, "Enable Safe, Scalable Parallelism with Intel Threading Building Block's Cocurrent Containers," DevX.com, Jupiter Media, December 2006, at http://www.devx.com/cplus/Article/33334.

AUTHORS' BIOGRAPHIES

Alexey Kukanov is a Senior Software Engineer in Intel's Performance Analysis and Threading Lab. Since joining Intel in 2000, he worked on a few software products. Finally, his interests in C++, library development and multi-threading have been happily combined in the TBB project where he is now one of leading developers. Alexey received an M.S. equivalent degree in Applied Math from Nizhny Novgorod State University. When he has some free time he likes playing billiards and volleyball. His e-mail is alexey.kukanov at intel.com.

Michael Voss is a Senior Staff Software Engineer in Intel's Performance Analysis and Threading Lab, where he is currently one of the lead developers of Intel Threading Building Blocks. Michael is also an adjunct professor in the Department of Electrical and Computer Engineering at the University of Toronto, where he taught from 2001–2005. His interests include languages, tools, and compilers for parallel computing. Michael received his Ph.D. and M.S.E.E degrees in Electrical Engineering from Purdue University in 2001 and 1997, respectively. His e-mail is michael.j.voss at intel.com.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4,

IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Intel's trademarks may be used publicly with permission only from Intel. Fair use of Intel's trademarks in advertising and promotion of Intel products requires proper acknowledgement.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Bluetooth is a trademark owned by its proprietor and used by Intel Corporation under license.

Intel Corporation uses the Palm OS® Ready mark under license from Palm, Inc.

Copyright © 2007 Intel Corporation. All rights reserved.

This publication was downloaded from
http://www.intel.com.

Additional legal notices at:
http://www.intel.com/sites/corporate/tradmarx.htm.

For further information visit:

developer.intel.com/technology/itj/index.htm

Copyright © 2007 Intel Corporation. All rights reserved.
Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.
For a complete listing of trademark information visit: www.intel.com/sites/corporate/tradmarx.htm