Journal of Information Processing Systems, Vol.8, No.2, June 2012 http://dx.doi.org/10.3745/JIPS.2012.8.2.331

TBBench: A Micro-Benchmark Suite for Intel Threading Building Blocks

Ami Marowka*

Abstract—Task-based programming is becoming the state-of-the-art method of choice for extracting the desired performance from multi-core chips. It expresses a program in terms of lightweight logical tasks rather than heavyweight threads.
Intel Threading Building Blocks (TBB) is a task-based parallel programming paradigm for multi-core processors. The performance gain of this paradigm depends to a great extent on the efficiency of its parallel constructs. The parallel overheads incurred by parallel constructs determine the ability for creating large-scale parallel programs, especially in the case of fine-grain parallelism.
This paper presents a study of TBB parallelization overheads. For this purpose, a TBB micro-benchmarks suite called TBBench has been developed. We use TBBench to evaluate the parallelization overheads of TBB on different multi-core machines and different compilers. We report in detail in this paper on the relative overheads and analyze the running results.

Keywords—TBB, Micro-Benchmarks, Multi-Core, Parallel Overhead

1. INTRODUCTION

A programmer expects that a parallel programming model will have three major properties: ease-of-use, high abstraction, and portable performance across a wide range of parallel architectures. The design of a parallel programming model that meets all of these expectations is still out of reach [2, 10-12, 21, 22, 24, 28]. Still, despite the many obstacles, there has been some progress in parallel programming that brings us closer to the desired parallel programming model [12].

Every HPC vendor is looking today for a breakthrough in this area. Over the past two decades, much effort has been invested in designing new parallel programming languages and models. For example, UPCRC [28] is a Microsoft-Intel-Academia initiative for finding new ways to program multi-core computers; Intel Parallel Studio is a new suite of parallel programming tools for the Microsoft Visual Studio environment [25]; and the Stanford parallel computing platform 2012 [1] from the Stanford Pervasive Parallelism Laboratory aims to make parallel programming practical for the masses. Most of the HPC research community proposals for new parallel programming paradigms did not attract HPC users because they did not offer reasonable solutions to many of the pitfalls and issues of multi-core programming [14, 15]. An exception is the Intel Threading Building Blocks (TBB) [17, 27].

Manuscript received July 21, 2011; first revision March 17, 2012; accepted April 9, 2012.
Corresponding Author: Ami Marowka
* Dept. of Computer Science, Bar-Ilan University, Israel (amimar2@yahoo.com)

Intel Threading Building Blocks (TBB) is a C++ template library that is aimed at developing parallel applications that run on top of multi-core processors. TBB designers are committed to making the TBB compiler, the processor, and operating system all independent. The library consists of building blocks (data structures and algorithms) that free a programmer from some complications arising from the use of native threading mechanisms such as threads creation, synchronization, and termination.

TBB abstracts access to the multiple processors by automating the process of data decomposition to small cache-friendly chunks of data processed by *tasks*, and then dynamically schedules the tasks to individual cores in an efficient manner. Tasks are blocks of code that perform a specific work or function when executed by the TBB runtime library. The motivation for using tasks rather than threads is the low overhead that is incurred when tasks are created and destroyed. Furthermore, tasks are dynamically assigned to available execution resources by the runtime library that helps to reduce load imbalance. This approach enables the programmer to focus on the program logic rather than on ways to use the underlying machine architecture efficiently. The core engine of TBB is the task scheduler, which uses a *task stealing* mechanism to balance a parallel workload across available processing cores in order to increase core utilization and therefore performance and scalability. After a TBB program is initialized, the task scheduler divides the workload among the threads to balance the load by allowing idle threads to steal tasks from the queue of busy threads.

The TBB library consists of algorithms (parallel for, parallel reduce, parallel scan, parallel while, pipeline, and parallel sort). The design of the algorithms is based on C++ generic programming and the recursively divisible ranges that are implemented on top of an efficient work-stealing scheduler. The library was designed for simplicity such that parallelism is mapped to the underlying machine resources without intervention by the programmer. In addition, the TBB library provides concurrent containers (concurrent queue, concurrent vector, and concurrent hash map). The containers are thread-safe and use fine-grained locking for efficiency. The library also contains concurrent memory allocation, various mutual exclusion mechanisms, and atomic operations.

However, as multi-core architectures continue to evolve they will require developers to refine their threading techniques as a core aspect of their solutions rather than as a merely desirable feature. Overheads associated with operations such as thread creation, synchronization and locking, threading granularity, scheduling, and process management will become more pronounced as time goes by, and the necessity of planning for parallel scalability will become more and more important. This paper makes the following contributions:

- We present the design methodology of the TBBench suite and show how to use its micro-benchmarks.
- We study the relative parallel overheads of TBB constructs on multi-core machines.
- We compare the parallel overheads of TBB on different multi-core architectures and on two different compilers, the Intel compiler and the Microsoft Visual Studio C++ 2008.
- We compare the parallel overheads of TBB and OpenMP on different multi-core machines.

Overall, the paper provides valuable insights that can help TBB users create more scalable and efficient parallel algorithms and applications. The rest of this paper is organized as follows: In Section 2, we introduce the design of TBBench and its components. Section 3 presents an in-depth analysis of the running results of TBB micro-benchmarks on different multi-core machines and compilers. In Section 4 we present related work, and Section 5 concludes the paper.

2. TBBENCH: A TBB MICRO-BENCHMARKS SUITE

TBBench is a suite of core benchmarks for measuring the overheads associated with the execution of TBB parallel constructs for synchronization, scheduling, and work-sharing. The design of TBBench follows the design concepts of the OpenMP EPCC micro-benchmarks suite.

The approach to measuring the overhead associated with a parallel construct is to compare the running time of a region of code running in parallel on *P* cores (*Tp*) to the running time of the same region of code running sequentially (*Ts*). The calculated overhead is given by taking the difference *Tp-Ts/P*. For example, by measuring the overhead associated with the TBB *parallel_for* construct, TBBench measures the time it takes to run the following region of code (*Tp*):

```
for (j =0; j<innerreps; j++){
parallel_for (blocked_range<int> (0, nthreads, 1), TBB_parallelforBody ());}
```

where TBB_parallelforBody () is the following class:

```
class TBB_parallelforBody {
public :
TBB_parallelforBody ( ) {}
/ / ! main loop
void operator ( ) ( const blocked_range<int> & range ) const {
for ( int i=range.begin ( ) ; i !=range.end();++ i ) {
delay (delaylengh ) ;
}}}
```

and then measuring the reference time (*Ts*), the time it takes to run the following loop on a single thread:

```
for ( j =0; j<innerreps ; j++){
delay ( delaylengh ) ;
}
```

And finally, subtracting the reference time divided by the number of cores (*Ts/P*) from *Tp*. Then, the result is divided by the number of iterations *innerreps* to get the overhead time per iteration.

TBBench is designed to pay attention to measuring statistically and reproducible results. The *delaylength* of the reference routine chosen is in the same order of magnitude as the parallel construct under evaluation but is large enough to guarantee that idle threads will start to steal tasks from busy threads to create parallel processing; the number of loop iterations *innerreps* chosen is larger than the clock resolution; and each running result reported here is an average of 20 different runs of 50 measurements each.

The micro-benchmark for measuring the overhead associated with the TBB *parallel_reduce* construct measures the time to run the following region of code:

```
Paralle_reduceBody b
for ( j =0; j<innereps j++){
parallel_redue ( blocked_range<int >(1 ,3 ,1) , b ) ;
}
```

where *parallel_reduceBody* is the following structure:

```
struct parallel_reduceBody {
int aaaa ;
/ / ! Constructor set aaaa to 0
Parallel_reduceBody ( ) : aaaa ( 0 ) {}
/ / ! Splitting constructor
Parallel_reduceBody ( parallel_reduceBody& 0ther , split ) : aaaa ( 0 ) {}
/ / ! Join point
void join ( parallel_reduceBody &s ) {
aaaa = aaaa + s . aaaa ;
}
void operator ( ) ( const blocked_range<int> &r ) {
for ( int k = r.begin( ) ; k < r.end ( ) ; ++k )
aaaa = aaaa + 1 ;
}}
```

Next, we demonstrate how TBBench measures the overhead incurred by TBB mutual exclusion mechanisms. The code below is the micro-benchmark for measuring the overhead of *Spin_mutex*. *Queuing_mutex* and *Mutex* mutual exclusion mechanisms have similar codes. The main routine invokes the call:

```
TBB_lock<spin_mutex>( ) ;
```

Where the TBB_lock is the following template that calls to the TBB parallel_for with the class parameter TBB_lockBody:

```
template<class M>
void TBB_lock ( )
{
M mutex ;
Parallel_for ( blocked_range<int >(0 , nthreads , 1 ) , TBB_lockBody<M>(mutex ) ) ;
}
```

The definition of TBB_lockBody is shown below and contains repeated calls to the TBBlock.acquire and the TBBlock.release pair:

```
template<typename M>
class TBB_lockBody {
M &mutex ;
public:
TBB_lockBody ( M &m ) : mutex( m ) {}
/ / ! main loop
void operator ( ) ( const blocked_range<int>& range ) const {
 typename M: : scoped_lock TBBlock ;
 for ( int i=range.begin( ) ; i !=range.end();++ i ) {
 for ( int j =0; j<innerreps / nthreads ; j++){
 TBBlock.acquire (mutex ) ;
 delay ( delaylength ) ;
 TBBlock.release ( ) ;
}}}}
```

Now, we will illustrate how TBBench benchmarks an ATOMIC mechanism. TBBench calls and measures the time of the following region of code:

```
atomic<int> aaaa ; aaaa=0;
parallel_for ( blocked_range<int >(0 , nthreads , 1 ) , TBB_atomicBody ( aaaa ) ) ;
```

Where TBB_atomicBody is the following class:

```
class TBB_atomicBody {
atomic<int> &paaaa ;
public:
TBB_atomicBody ( atomic<int> &m) : paaaa (m) {}
/ / ! main loop
void operator ( ) ( const blocked_range<int>& range ) const {
 for ( int i=range.begin( ) ; i !=range.end();++ i ) {
 for ( int j =0; j<innerreps / nthreads ; j++){
  paaaa.fetch_and_add ( 1 ) ;
}}}}
```

Finally, the code for measuring the scheduling overhead is presented. For this purpose TBBench invokes and measures the time of the following region of code:

```
parallel_for ( blocked_range<int >(0 , itersperthr * nthreads , cksz ) ,
 TBB_schedBody ( ) ) ;
```

The variables *itersperthr, nthreads,* and *cksz* refer to the number-of-iterations per thread, the number-of-threads, and the chunk-size respectively. The TBB_schedBody is the following class:

```
class TBB_schedBody {
public:
TBB_schedBody ( ) {}
/ / ! main loop
void operator ( ) ( const blocked_range<int>& range ) const {
 for ( int i=range.begin( ) ; i !=range.end();++ i ) {
 for ( int j =0; j<innerreps ; j++){
  delay ( delaylength ) ; }}}}
```

The schedule options *simple_partitioner*, which recursively splits a range until it is no longer divisible, and *auto_partitioner*, which guides splitting decisions based on the work-stealing behavior of the task scheduler, are also measured by calling to the following codes respectively:

```
parallel_for ( blocked_range<int >(0 , itersperthr*nthreads ) , TBB_schedBody ( ) , sim-
ple_partitioner ( ) ) ;
```

```
parallel_for ( blocked_range<int >(0 , itersperthr*nthreads ) , TBB_schedBody ( ) ,
auto_partitioner ( ) ) ;
```

3. Experimental Results

This section describes and analyzes the benchmarks performed to evaluate the parallelization overheads of different TBB parallel constructs by using TBBench. The analysis of the running results examines the parallel overheads from different angles. We compare the parallel overheads of different TBB constructs on various multi-core processor architectures.

The list of the tested platforms is shown in Table 1. On the software side we used TBB versions 2.2 under the Intel C++ compiler 11.0 for Windows and Microsoft Visual Studio C++ 2008. Both compilers were run on top of the XP operating system. All of the measurements shown in the figures below are in Kilo-Clock-Cycles for providing a better comparison between different machine architectures.

TBBench cannot be considered a "black box" benchmarking tool. In other words, this tool depends on the settings and the tunings of a few parameters that affect the accuracy and the reproducibility of the measurements. There are external parameters such as the number of threads and compiler options (optimization level, runtime library type, etc.) and there are internal parameters such as *delaylength, innerreps,* and *itersperthr* (See section 3). Moreover, attention has to be paid to the fluctuations of the measurements in order to achieve statistically stable and reproducible results. There are many reasons for getting different measurements from run to run such as the non-deterministic behavior of the scheduler, different allocations of variables in the memory space, the accuracy of time measurements of the clock routines, unpredictable context switches, initialization overheads, and the inconsistency of the cache memories' behavior. Similar problems were reported in [3, 4].

One important lesson that we learned from this work is that analysis of the measurement results sometimes yield unexpected behaviors. Some of these behaviors can be explained by using profiling and performance tools that have access to the hardware counters. Unfortunately, there are cases that cannot be explained without knowing the specific implementation of the tested software and only educated guesses can be given to explain the unexpected results [16].

The default compiler option used was /O2 (maximize speed). The internal parameters of TBBench were set to *delaylength* = 500 (the granularity of loop body reference), *innerreps* = 10000 (the number of loop iterations) and *itersperthr* =128 (the number of iterations per thread for the scheduling measurements). The benchmarking results reported here are averages of 20 runs of 50 measurements each for statistical stability. Any measurement that exceeded the threshold of three times the standard-deviation was ignored.

Table 1. The tested multi-core machines

Platform	No. of Cores	Clock(GHz)	L1 Cache	L2 Cache	L3 Cache	Memory
Intel Pentium D 820	2	2.8	2x16KB	2x1MB	-	1GB
Intel Core 2 Duo T8100	2	2.1	2x32KB	1x3MB	-	1GB
Intel Core 2 Quad Q6600	4	2.4	4x32KB	2x2MB	-	1GB
AMD Athlon X2 7750	2	2.7	2x64KB	1x1MB	-	1GB
AMD Phenom X4 9750	4	2.4	4x64KB	4x512KB	1x2MB	1GB

3.1 Work-sharing benchmarks

Fig. 1 plots the bar charts of the parallel overheads incurred by TBB work-sharing constructs. The measurements shown are of *Parallel_For* and *Parallel_Reduce* for two threads in the Intel

Core 2 Duo machine. Since TBB designers are committed to making the TBB compiler, processor, and operating system all independent we examined the results for two compiler options (/O2 for maximize speed and /Od for optimization disabled) and two different compilers (Intel and Microsoft).

First, it can be observed from Fig. 1 that optimization can improve performance by up to 23% in the case of *Parallel_for* and the Intel compiler. Second, the Intel compiler achieves better performance (up to 4% for the /O2 case) for both work-sharing constructs, while the Microsoft compiler achieves better performance (up to 5%) when the optimization is disabled. These observations suggest that although TBB work-sharing construct implementations were not designed for a specific compiler, different compilers can exhibit different performances and compiler optimizations can reduce overheads.

Fig. 2 plots the bar charts of the parallel overheads incurred by TBB work-sharing constructs on five different multi-core architectures and makes a comparison between TBB and OpenMP parallel overheads. The OpenMP parallel overheads were obtained by using the EPCC microbenchmark suite. The results shown in Fig. 2 were obtained by using the Intel compiler with its maximize speed (/O2) compiler option enabled and by using two threads.

It can be observed that OpenMP outperforms TBB for any kind of multi-core architecture. OpenMP achieves an up to eight times better performance in the case of *Parallel_For* on the oldest architecture in Fig. 1 (Pentium D) and performs up to three times better in the case of *Parallel_Reduce* on the newest architecture (Phenom X4). These findings suggest that OpenMP implementations of these work-sharing constructs are more efficient because OpenMP is not a compiler independent programming model and thus can be better optimized by the compiler.

Fig. 2 illustrates the impact of different multi-core architectures on TBB parallel overheads. It can be observed that the newest architectures significantly reduce the parallel overheads as compared to former architectures. This may largely be due to improvements in the underlying hardware, especially in the memory subsystems. Note that TBB parallel overheads are more influenced by the underlying architecture as compared to OpenMP. This is because TBB is a compiler independent library and thus future improvements in the underlying architectures can significantly improve the performance of TBB applications.

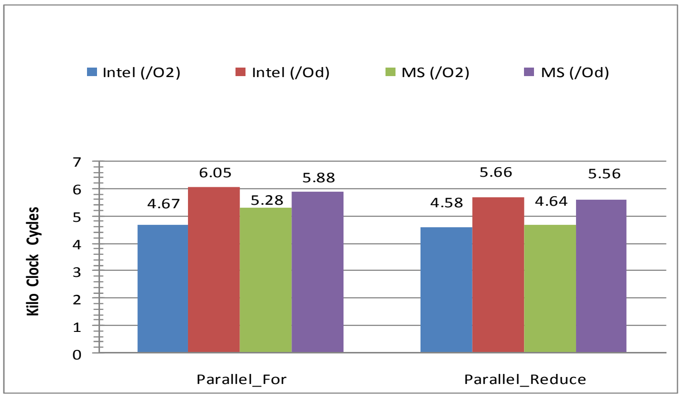

Fig. 1. Parallel overheads of TBB Parallel_For and Parallel_Reduce operations for 2 threads in Core 2 Duo, the T8100, 2.1 GHz machine, and for Intel and Microsoft compilers with their maximize speed (/O2) and optimization disabled (/Od) compiler options

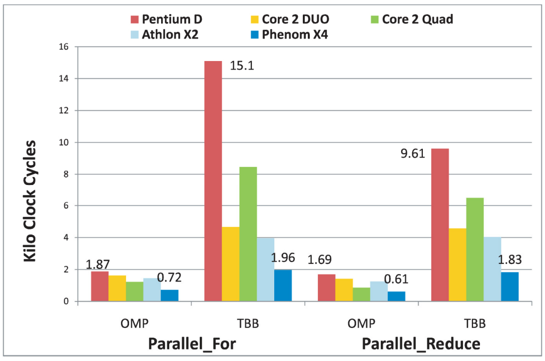

Fig. 2. Parallel overheads of Parallel_For and Parallel_Reduce operations of OpenMP vs. TBB for 2 threads in different multi-core machines, and for the Intel compiler with the maximize speed (/O2) compiler option

3.2 Mutual exclusions benchmarks

Fig. 3 plots the bar charts of the overheads incurred by TBB mutual exclusion mechanisms. The measurements shown are *Mutex, Queuing Mutex, Spin Mutex* and *Atomic* for two threads on the Intel Core 2 Duo machine. Again, since TBB designers are committed to making the TBB compiler, processor, and operating system all independent we examined the results for two compiler options (/O2 for maximize speed and /Od for optimization disabled) and two different compilers (Intel and Microsoft).

A *Mutex* in TBB is a global variable that multiple tasks can access. Protecting a *Mutex* is done by locking mechanisms. *Spin_Mutex* causes a task to *spin* in user space while it is waiting and Mutex causes a task to sleep. For short waits, spinning in user space is fastest because putting a task to sleep takes cycles. *Queuing_Mutex*, like *Spin_Mutex,* causes a task to spin but each task gets its turn based on a First-In First-Out (FIFO) policy.

The principal observation from these results is that *Mutex* and *Queuing_Mutex* incur substantial overheads in comparison to *Spin_Mutex*. This appears to be largely due to the cost of managing these mutexes. As expected, the *Spin_Mutex* presents the lowest overhead and it is most

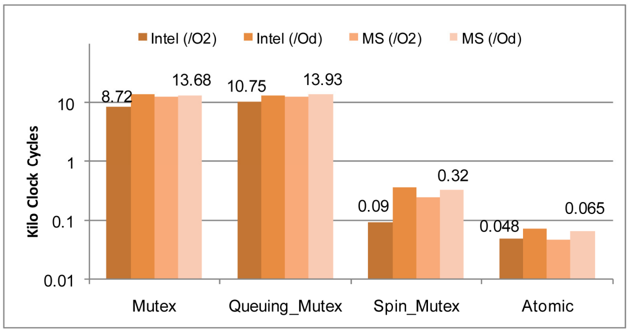

Fig. 3. Parallel overheads of TBB mutual exclusions operations Mutex, Queuing_Mutex, Spin_ Mutex, and Atomic operations for 2 threads on Core 2 Duo, T8100, the 2.1 GHz machine, and for Intel and Microsoft compilers with maximize speed (/O2) and optimization disabled (/Od) compiler options

likely its simple implementation that makes *Spin_Mutex* very fast in lightly contended situations such as ours, where the benchmarks are executed with only two threads. Therefore, we omit the discussion on scalability with respect to the number of cores in this article because it is useless to do such an analysis when the machines have only two or four cores. Fig. 3 shows that the Intel compiler achieves better performance than the Microsoft compiler for the three mutex mechanisms (up to three times better for the optimized case of *Spin_Mutex*), while the Microsoft compiler achieves better performance (up to 6% better for the optimized case) as compared to the Intel compiler for the Atomic operation. The TBB Atomic operation achieves a low and very stable overhead, as is expected from an operation that just call for a low level operation such as *fetch_and_add*. Simulations show that as the number of worker threads is increased, atomic operations can become a significant source of performance degradation when a relatively large number of tasks are created [5]. Fig. 4 compares the parallel overheads of OpenMP and TBB with respect to the *Lock_Unlock* and *Atomic* operations on five multi-core architectures with an Intel compiler. The *Lock_Unlock* comparison in Fig. 4 refers to the *Lock* and *Unlock* routines of

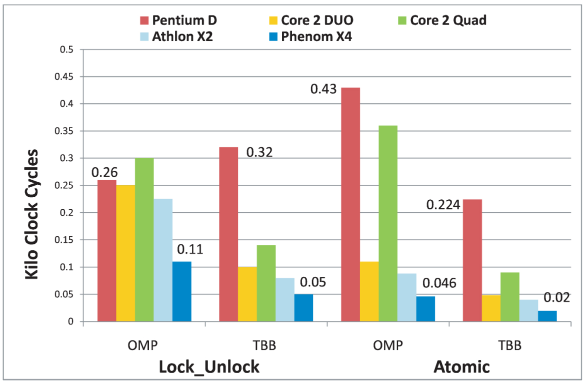

Fig. 4. Parallel overheads of Lock_Unlock and Atomic operations of OpenMP vs. TBB for 2 threads on different multi-core machines, and for an Intel compiler with the maximize speed (/O2) compiler option

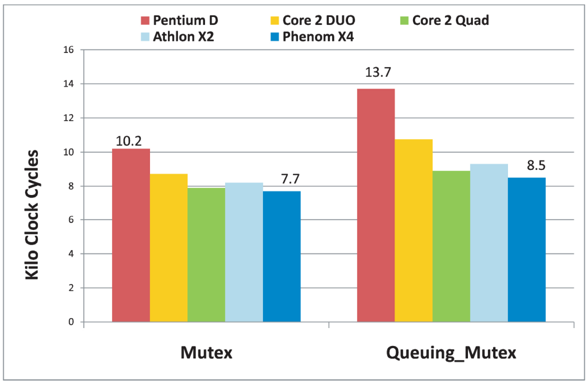

Fig. 5. Parallel overheads of TBB Mutex and Queuing_Mutex operations for 2 threads on different multi-core machines, and for an Intel compiler with the maximize speed (/O2) compiler option

OpenMP and to the *Spin_Mutex* of TBB. The OpenMP *Lock_Unlock* and TBB *Spin_Mutex* have the same semantic. It can be observed that TBB appears to achieve less overhead than OpenMP on four of five architectures. The exception is Pentium D, the oldest architecture on the list, which exhibits relative high overheads in all the cases. This is most likely due to the inefficient implementations of OpenMP mutual exclusion mechanisms, which suggest that an Intel compiler can significantly improve the OpenMP implementations of these mechanisms. Fig. 5 depicts the parallel overheads of TBB *Mutex* and *Queuing_Mutex* on the five tested architectures with an Intel compiler (with /O2 compiler option enabled) and multi-core machines with two cores and two threads. Note that these two mutexes are very expensive for all the architectures as compared to *Spin_Mutex* and thus it is recommended to use them only when it is necessary.

3.3 Scheduling benchmarks

Parallel overheads for different chunk sizes of TBB schedules are given in Fig. 6. The running results shown in Fig. 6 were obtained by running TBBench on an Intel Core 2 Duo machine with two threads and two compilers (Intel and Microsoft) for 1, 2, 4, 8, 16, 32, 64, and 128 chunk sizes, and for the scheduling options *auto_partitioner* and *simple_partitioner*.

First, it can be observed that TBB consumes less overhead (up to 2.25 times less for a chunk size of 64) with the Intel compiler. However, there is one exception where the Microsoft compiler achieves better performance (for a chunk size of 2) but the reason why is not clear. Moreover, the *Simple* option of the TBB scheduler outperforms any manual setting of a chunk size in both compilers. *Simple* refers to TBB's *simple_partitioner* option that recursively splits a range until it is no longer divisible and *Auto* refers to TBB's *auto_partitioner* option that guides splitting decisions based on the work-stealing behavior of the task scheduler. Fig. 7 presents the behavior of the scheduling options *simple_partitioner* and *auto_partitioner* for our tested platforms. Again, it can be observed that the oldest architecture (Pentium D) obtained a relatively high overhead as compared to the newest architectures.

Fig. 8 plots the bar charts of the scheduling overheads incurred by TBB vs. OpenMP (static)

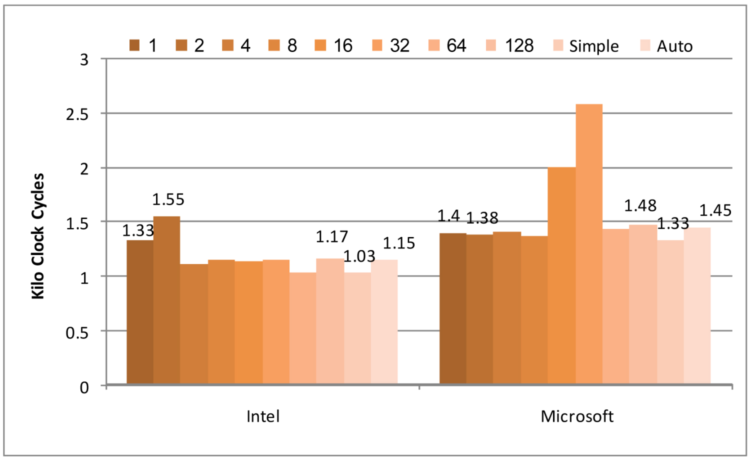

Fig. 6. Scheduling overheads of TBB for the Intel compiler vs. the Microsoft compiler for 2 threads on core 2 Duo, T8100, and the 2.1 GHz machine with the maximize speed (/O2) compiler option

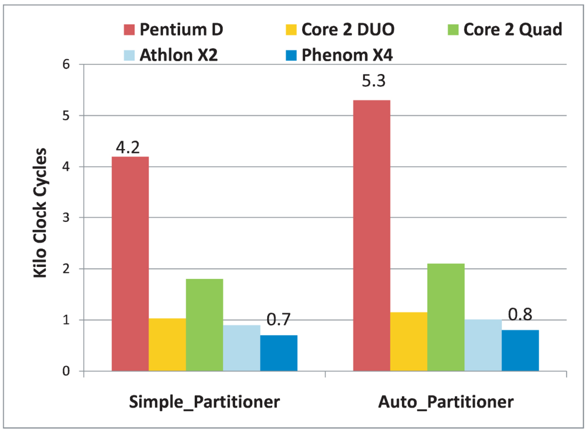

Fig. 7. Scheduling overheads of TBB Simple_Partitioner and Auto_Partitioner options for 2 threads on different multi-core machines, and for Intel compiler with maximize speed (/O2) compiler option

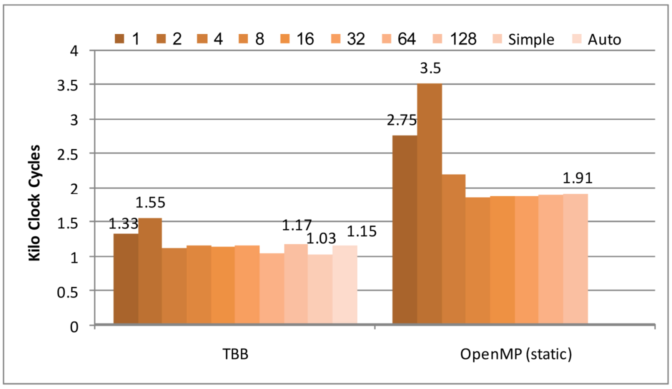

Fig. 8. Scheduling overheads of OpenMP(static) vs.TBB for 2 threads on Core 2 Duo,T8100, 2.1 GHz machine, and for an Intel Compiler with the maximize speed (/O2) compiler option

for two threads on the Intel Core 2 Duo machine and the Intel compiler, with /O2 option enabled. We chose to compare TBB against the static schedule of OpenMP because it is usually the most efficient choice [13]. It can be observed from Fig. 8 that TBB outperforms OpenMP (approximately two times better). Moreover, the *simple_partitioner* and *auto_partitioner* TBB scheduler options exhibit a better performance than any OpenMP manual setting. This observation suggests that the recursive-based TBB scheduler is more efficient than the static scheduler of OpenMP.

4. Related Work

The EPCC micro-benchmarks for OpenMP were introduced by Bull in [3]. In his introduction, the author explains the design methodology of the benchmarks and then offers an evaluation of the synchronization and loop scheduling overheads incurred by OpenMP on three different platforms (SGI Origin 2000, SUN HPC 3500 and Compaq Alpha server). The author emphasizes that particular attention was paid to derive statistically stable and reproducible results. However, the author's discussion on the benchmark results does not mention whether the goals of statistical stability and reproducibility were achieved. The measurements were conducted while the number of iterations per thread was fixed to 1,024 and the granularity of the loop body was tuned to approximately 100 cycles. Moreover, each overhead measurement was repeated 50 times per run for 20 runs because of the variability of the measurements from run to run. The reasons for the high fluctuations in the measurements are not clear to the author who surmises that it happens due to alterations of the memory locations of the synchronization variables from run to run. An analysis of the results leaves the reader with more doubts about the ability to measure accurate results. For example, the measurements of the LOCK/UNLOCK functions and DYNAMIC loop scheduling were not obtained due to frequent deadlocks. Moreover, the performance of a SINGLE directive on the Sun machine showed less overhead than a BARRIER directive, although a SINGLE directive contains an implicit BARRIER operation.

In [4] Bull and O'Neill extensions are presented to the EPCC micro-benchmarks for OpenMP version 2.0 that measures the overhead of WORKSHARE, PARALLEL WORKSHARE directives, PRIVATE, FIRSTPRIVATE, and COPYIN clauses. The new extensions were benchmarked on Sun HPC 6500 and SGI Origin 3000 machines. The authors explain that there is a conflict between increasing the size of the WORKSHARE array (and thus spending less time in the directive) and decreasing the size of the array and then these arrays will be vulnerable to false-sharing situations. This conflict demonstrates the difficulties of producing accurate overhead measurements.

Contreras and Martonosi studied the basic parallelism management costs of the TBB runtime Library [5]. On the hardware side, their testing platform was a quad-core machine for measuring performance on 1-4 real cores and simulations for studying the overheads on 4-32 virtual cores. On the software side, they used four micro-applications of the PARSEC benchmark suite (*Fluidanimate, Swaptions, Blackscholes*, and *Streamcluster*) that were ported to TBB, and four kernel-benchmarks (*Bitcounter, Matmult, LU* and *Treeadd*). In measuring the basic operations of the TBB runtime library they focused on five common operations: *Spawn, Get_task, Steal, Acquire_queue,* and *Wait_for_all.*

Analysis of the benchmarking results reveals the following findings: synchronization overheads within TBB have a significant impact on parallelism performance; the runtime library contributes up to 47% of the total per-core execution time on a 32-core system (due to synchronization overhead within the TBB scheduler) and hinders performance scalability; the random task-stealing mechanism becomes less effective as application heterogeneity and core counts increase; and a queue occupancy-based stealing policy can improve the performance of task stealing by up to 17%.

The work of Contreras and Martonosi provides important findings for programmers and TBB designers. However, important information is missing regarding their measurements for providing better evaluation of their findings and observations. The authors did not report how much attention was paid to deriving statistically stable and reproducible results. Moreover, the results show significant differences between the simulation pattern results and hardware (non-simulation) pattern results. It is not clear whether the differences are due the differences of the tested architectures or to the software implementations and thus creates some doubts about the accuracy of the measurements and their ability to assess the scalability of TBB on manycore processors.

Sphinx is an integrated parallel micro-benchmark suite for evaluating the performance of MPI, Pthreads, and OpenMP programming models [26]. Sphinx-OpenMP uses similar methodology measurements to those used in the EPCC micro-benchmark suite. Sphinx measures several timings and outputs for their arithmetic means for a given set of parameters for the action. The timings are stopped when the standard deviation of the repetitions is less than a user-defined threshold, given that a minimum number of repetitions have been measured. Since this cut-off may never be achieved, Sphinx guarantees test termination through a user-specified maximum number of repetitions.

OmpP [6-8] is a Linux-based OpenMP profiler. OmpP presents the profiling results in a text-based manner but gives the user a fast user-friendly report about hotspots in the tested application. OmpP supports the measurement of hardware performance counters by using the PAPI library where HWC events are selected via environment-variable settings. The analysis report categorizes the overhead into four types: imbalance, synchronization, limited parallelism, and thread management. The detailed analysis helps the programmer detect common bottlenecks that inhibit achieving the desired scalability, and discovers inefficiencies that are subjects for further in-depth investigation.

Wang and Xu [23] studied the scalability of the multiple-pattern matching algorithm known as the Aho-Corasick-Boyer-Moore Algorithm on the Intel Core 2 Duo processor 6300, 1.86 GHz with 1GB main memory, and on Windows XP for different input sizes. This work compares the performance of the Windows Threading API with the Intel Threading Building Blocks. The authors found that the average scalability achieved by TBB is 1.655 as compared to 1.549 of Win32. The authors explain that TBB achieves better performance because it specifies tasks instead of threads. A task can be dynamically assigned to a thread. Intel TBB selects the best thread for a task by using the task scheduler. If one thread runs faster, it is assigned to perform more tasks. With the Win32 Threading Library, however, a thread is assigned to a fixed task and cannot be reassigned to other tasks even if it is idle.

Robison et al. [20] studied two different optimization strategies that aim to improve the performance of the work-stealing task scheduler of TBB. The first optimization automatically tunes the grain size based on inspection of the stealing behavior. The second optimization improves cache affinity by biased stealing. For testing the impact of the grain size on the performance, the authors used *Pi*, a simple numeric integration benchmark that computes π. The results show that OpenMP *static* achieves the best speedup as compared to all OpenMP and TBB scheduling strategies, while the TBB *affinity_partitione* achieves the best speedup as compared to other TBB strategies. The cache affinity benchmark used two programs: Sesmic, which is a program that simulates 2D wave propagation and Cholesky, which is a program that decomposes a dense symmetric square array A into the lower triangular matrix L such that $A = LxL^T$. Only OpenMP *static* scheduling and TBB *affinity_partitioner* gave good results for more than two threads. OpenMP's static scheduling did the best.

Kegel et al. [18] used different OpenMP and TBB implementations of a block-iterative algorithm for 3D image reconstruction (ListMode Ordered Subset Expectation Maximization) for comparison between OpenMP and TBB with respect to programming effort, programming style, and performance gain. The authors studied five implementations (two OpenMP and three TBB) that differed in the locking mechanisms that they used for synchronization. One of the OpenMP implementations used the atomic mutual exclusion mechanism while the second one used the critical mechanism. The three TBB implementations use mutex, queuing mutex, and spin mutex mechanisms respectively.

The authors found that OpenMP offers a more simple and easy way to parallelize existing sequential code as compared to TBB, which demands for the redesign and rewriting of the sequential code and is thus more appropriate for building parallel programs from scratch. Moreover, although the two programming paradigms support high-abstraction APIs, OpenMP outperforms TBB while achieving better scalability due to more efficient locking mechanisms.

Podobas et al. [19] studied the performance results of five applications (FFT, NQueens, Multisort SparseLU, and Strassen) with six implementations of task-parallel programming models (four implementations of OpenMP, Cilk++ and Wool). The OpenMP implementations used four different compilers (GCC, Intel, Sun, and Mercurium). On the low-level, they studied the costs of task-creation and task-stealing.

The authors found that Wool and Cilk++ achieve low overheads in task-spawning and task-stealing as compared to OpenMP. Among the four implementations of OpenMP, the Intel compiler exhibited the lowest overheads. However, they measured relatively high overheads with respect to the parallel constructs of the tested applications. Even so, for coarse-grained tasks all the applications achieved high speedups. The authors said that most of their findings are still not clear and need further investigation.

Hower and Jackson [9] offer a C++ library called TaskMan that realizes the task-based programming paradigm. The library makes use of futures to introduce a call/return API that is usually found in imperative languages. The primary designing goals of TaskMan are simplicity and programmability. A performance study shows that TaskMan does not achieve the performance level of TBB and Cilk++ but has the potential to close the gap while delivering a lightweight and intuitive API.

5. Conclusions

The course of change in the hardware industry is clear - the primary means of evolution for processors in the foreseeable future is through increasingly large numbers of execution cores per processor package. Understanding their performance characteristics is essential for designing scalable and efficient applications. In this paper, we introduced TBBench, a micro-benchmark suite for TBB that we used to measure the parallel overhead costs of TBB parallel constructs. The benchmarks were conducted on five multi-core generations with two different compilers.

The benchmarking results show that the Intel compiler usually achieves better performance as compared to the Microsoft compiler, while the /O2 compiler option for optimized speed improves the performance even further. Moreover, the implementations of the OpenMP work-sharing constructs of the Intel compiler are efficient and outperform the equivalent TBB constructs. On the other hand, TBB mutual exclusion mechanisms exhibit less overhead in comparison to their equivalent OpenMP constructs. The scheduling benchmarks show that TBB achieves better performance as compared to OpenMP due to its sophisticated scheduler while the TBB schedule option *simple_partitioner* exhibits the best performance. The benchmarks were conducted on five different machine architectures and showed that improvements of the memory subsystems lead to better utilization of the hardware resources and reduce parallel overheads.

APPENDIX

TBBench micro-benchmark suite can be downloaded from:
http://amimarowka.weebly.com/tbbench.html

The version 1.0 of the benchmark suite is a self-contained C/C++ file.
TBB is an open-source package that can be downloaded from:
http://threadingbuildingblocks.org/

There are a few parameters that must be set:

- **MHz** - the CPU rate in Mega Hertz.
- **Innerreps** - the number of loop iterations. This is chosen to be larger than the clock resolution.
- **Delaylength** - the delay duration of the reference routine. It is chosen to be in the same order of magnitude as the parallel construct under evaluation but it is large enough to guarantee that idle threads will start to steal tasks from busy threads for creating parallel processing.
- **Itersperthr** - number-of-iterations per thread. (for the scheduling benchmark)
- **CKSZ** - chunk-size. (for the scheduling benchmark)

REFERENCES

[1] A. Aiken et al., "*Towards Pervasive Parallelism*". Presentation of Pervasive Parallelism Laboratory Stanford University, http://ppl.stanford.edu/wiki/index.php/Pervasive_Parallelism_Laboratory.

[2] K. Asanovic et al., "*The landscape of parallel computing research: A view from Berkeley*". University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December, 18, 2006. http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[3] M. Bull, "*Measuring Synchronization and Scheduling Overheads in OpenMP*", *Proceeding of First European Workshop on OpenMP (EWOMP '99) Lund*, Sweden, October, 1999.

[4] M. Bull and D. O'Neill, "*micro-benchmark Suite for OpenMP 2.0*", *Proceedings of the Third European Workshop on OpenMP (EWOMP'01)*, Barcelona, Spain, September, 2001, pp.41-48.

[5] G. Contreras and M. Martonosi, "*Characterizing and Improving the Performance of Intel Threading Building Blocks*", *IEEE Proceeding of International Symposium on Workload Characterization*, 2008, pp.57-66.

[6] K. Fuerlinger and M. Gerndt, "*ompP: A profiling tool for OpenMP*", In Proceedings of *the First International Workshop on OpenMP (IWOMP 2005)*, Eugene, Oregon, USA, May, 2005.

[7] K. Fuerlinger, "*The OpenMP Profiler ompP: User Guide and Manual*", May, 2008. http://www.cs.utk.edu/ karl/research/ompp/usage.html

[8] K. Fuerlinger and D. Skinner, "*Performance Profiling for OpenMP Tasks*", In Proceedings of *the 5th International Workshop on OpenMP (IWOMP 2009).* Dresden, Germany, June, 2009.

[9] D. Hower and S. Jackson, "*TaskMan: Simple Task-Parallel Programming*", http://pages.cs.wisc.edu/ david/courses/cs758/Fall2009/includes/Projects/JacksonHower-slides.pdf

[10] B. Nicols et al., "*Pthreads Programming, A POSIX Standard for Better Multiprocessing*", O'reilly, September 1996.

[11] A. Marowka, "*Parallel Computing on Any Desktop*", Communication of ACM, Vol.50, Issue 9, September, 2007, pp.74-78.

[12] A. Marowka, "*Execution Model of Three Parallel Languages: OpenMP, UPC and CAF*". Scientific Programming, Vol.13(2), October, 2005, pp.127-135.

[13] A. Marowka, "*Performance of OpenMP Benchmarks on Multi-core Processors*", *8th International Conference on Algorithms and Architectures for Parallel Processing(ICA3PP)*, Agia Napa, Cyprus, June, 9-11, 2008, LNCS proceeding Vol.5022, pp.208-219.

[14] A. Marowka, "*Pitfalls and Issues of Manycore Programming*", ADVANCES IN COMPUTERS, Volume 79, 2010, Elsevier.

[15] A. Marowka, "*Back to Thin-Core Massively Parallel Processors*", IEEE Computer, Vol.44, No.12, December, 2011, pp.49-54.

[16] A. Marowka, "*On Performance Analysis of a Multithreaded Application Parallelized by Different Programming Models using Intel VTune*", Malyshkin, V. (ed.) *Eleventh International Conference on Parallel Computing Technologies (PaCT)*. LNCS 6873, Springer (2011), pp.317-331.

[17] J. Reinders, "*Intel Threading Building Blocks, Outfitting C++ for Multi-core Processor Parallelism*", O'Reilly, 2007.

[18] P. Kegel, M. Schellmann, S. Gorlatch, S. (2009): "Using *OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-Cores*". In Sips, H. J., Epema, D. H. J., Lin H. (Hrsg.): Euro-Par 2009 Parallel Processing, *15th International Euro-Par Conference, Delft*, The Netherlands, August, 25-28, 2009, Seiten 654-665.

[19] A. Podobas, M. Brorsson, and K. Faxan, "*A Comparison of some recent Task-based Parallel Programming Models*", in the proceeding of *the Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, Pisa, Italy, January, 24, 2010.

[20] A. Robison, M. Voss and A. Kukanov, "*Optimization via Reflection on Work Stealing in TBB*", In Proceeding of IEEE *International Symposium on Parallel and Distributed Processing, IPDPS*, 2008, pp.1-8.

[21] H. Sutter, "*The free lunch is over: A fundamental turn toward concurrency in software*". Dr. Dobb's Journal, 30(3), March, 2005.

[22] H. Sutter and J. Larus, "*Software and the concurrency revolution*". ACM Queue 3, 7 (September, 2005), 54-62.

[23] L. Wang and X. Xu, "*Parallel Software Development with Intel Threading Analysis Tools*", Intel Technology Journal, Vol.11, Issue 04, 2007, pp.287-297.

[24] "*High Productivity Computing Systems*", http://www.highproductivity.org/

[25] "*Intel Parallel Studio*", http://www.intel.com/cd/software/products/asmo-na/eng/399359.htm

[26] "*Sphinx Micro-benchmark Suite*", http://www.llnl.gov/CASC/RTS Report/sphinx.html

[27] TBB Web Site: http://www.threadingbuildingblocks.org/

[28] UPCRC: http://www.upcrc.illinois.edu/index.html

Ami Marowka

He is an adjunct Assistant Professor at the Computer Science Department of Bar-Ilan University in Israel. His research interests include the portability of HPC applications, parallel computing, the use of advanced-computer architectures, programming methodology, and tools for parallel computers. He received a PhD in Computer Science from the School of Computer Science and Engineering at the Hebrew University in Jerusalem, Israel.