Improving Blocking Operation Support in Intel TBB

Jiri Dokulil, Siegfried Benkner, and Martin Sandrieser
Research Group Scientific Computing
University of Vienna, Austria
Email: jiri.dokulil@univie.ac.at

***Abstract*—The Intel Threading Building Blocks (TBB) template library has become a popular tool for programming many-core systems. However, it is not suitable in situations where a large number of potentially blocking calls has to be made to handle long-running operations like disk access or remote data access. We have designed and implemented an add-on for the TBB that allows developers to better integrate long-running operations into their applications. We have extended TBB's task dependencies to also include blocking operations and implemented a run-time that efficiently manages these dependencies.**

I. Introduction

The Intel Threading Building Blocks (TBB) C++ template library [1] has become a popular tool for programming many-core systems. Based on work-stealing techniques, it enables efficient parallel execution of a set of computational tasks with dependencies. The library is suitable for a wide range of applications and enables even mainstream developers to utilize the power of multi-core systems.

However, the use of blocking operations within TBB tasks is not advised and it should be avoided whenever possible. Despite that, there are situations where it is necessary to wait for completion of potentially blocking operations. Examples for such situations could be I/O tasks such as reading a file or communication over a network.

Another situation where blocking may be required is when a conventional general-purpose CPU is used in conjunction with accelerators or coprocessors specialized in highly parallel execution or specific workloads. Examples of such *heterogeneous computing* include the utilization of GPUs via CUDA [2] or OpenCL [3], or using highly parallel Intel Xeon Phi coprocessors [4]. Programming such heterogeneous systems is often based on an *offloading* model where a general-purpose host-CPU offloads data and computations to an attached accelerator or coprocessor. Offloading usually requires executing operations on the host that wait for the completion of data transfers or computations on the accelerator or coprocessor.

We have developed the HyPHI heterogeneous programming library for Intel Xeon Phi that utilizes many TBB programming concepts [5]. This library automates offloading of computational tasks to multiple Xeon Phi coprocessors while at the same time utilizing all of the CPUs on the host for computations. The host-portion of such fully-*hybrid* execution can suffer from significant performance degradation caused by blocking operations associated with data-management and task execution for the coprocessors. Since we built our library on TBB tasks, we wanted to implement a convenient and efficient method to support the required blocking operations.

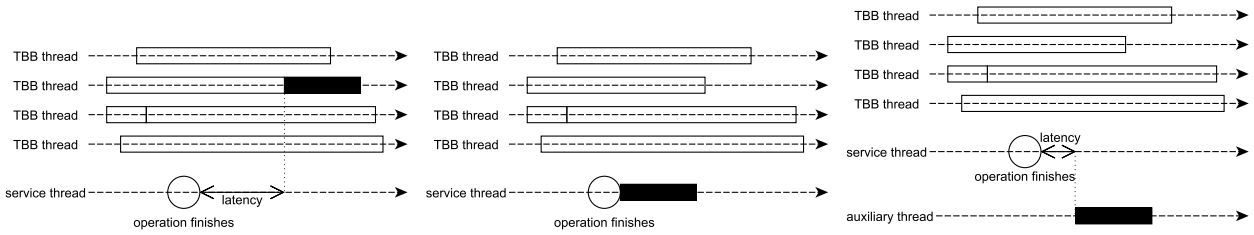
Even though many current programming frameworks allow potentially blocking operations to be executed asynchronously, querying for operation completion and possible overlap with other work has to be implemented by the user inside a task, if a pure task-parallel environment is used. When external task-based libraries are used, this may significantly affect task-scheduling performance since the framework might not be aware of the blocking calls made by the tasks. We believe, that it can be more beneficial to describe blocking operations more explicitly, to make the task scheduler aware of them.

In this paper, we present a TBB extension (add-on) designed to support blocking operations. Our approach is based on an extension of the TBB task graph. Instead of blocking TBB tasks, we utilize a separate *service thread* for asynchronous execution of waiting operations. This external mechanism interacts with TBB by spawning successor tasks of blocking operations as soon as their dependencies are fulfilled. Hence, the blocking operations effectively become new nodes in the TBB task graph and the new operation-task dependencies are handled in a way that is consistent with normal TBB task-task dependencies. We show that even though our approach relies on an additional thread, its overhead is negligible. In addition, we provide different strategies that allow the user to select whether it is more important to get an immediate response to a finished blocking call or to reduce the total overhead caused by blocking calls.

This paper is structured as follows: Section II provides an introduction to the problem and the main points of our solution. Section III discusses the application of the proposed concepts within the HyPHI library. Section IV provides more detailed information about our work, first showing the architecture of our software in the context of TBB. Then, we point out relevant details of the TBB implementation and show the way they are used to implement our extension. Section V investigates the latency encountered when spawning successor tasks and shows the different strategies we provide to deal with it. Experimental results are presented in Section VI. Related work is discussed in Section VII. Section VIII concludes the paper.

II. Problem and solution overview

There are situations, where a program needs some data to continue, but the data is not immediately available and it cannot be generated by the CPU. An example may be reading a file from disk, since the disk is not able to keep up with the performance of the CPU and loading a file takes extremely



(a) Normal – task is submitted to TBB (b) Short – executed by the service thread (c) ASAP – executed by an aux. thread

Fig. 1: Different strategies used to start a task that follows a blocking operation. The circle marks the time when the asynchronous operation finishes (`operation1` ends), the black bar represents the task that depends on the operation (`task3`).

long from the perspective of a CPU. Another example may be a remote procedure call or another kind of network operation. In such situations, the network latency is combined with the time required by the remote machine to process the request. A similar case is transferring data or executing a computational kernel on an accelerator (like a GPU) or a coprocessor (Intel Xeon Phi). An interesting property of such actions is that they often can run asynchronously. That is, once the CPU issues the command, it can move on to do something else and the remote command keeps executing (within operating system, hard-drive hardware, network hardware, remote machine etc.).

There are several ways in which the process that issued the asynchronous command can be made aware of its completion. These include callbacks, polling, and blocking calls. Callback is a function that gets called once the operation finishes. Generally, callbacks require careful design and implementation to avoid race conditions. Polling means that the waiting process has to keep checking whether a flag has been set or not. If they test the flag often, it is a waste of resources, if they test the flag rarely, there may be a significant delay between the time the operation finishes and the moment the process becomes aware of this fact. Blocking call is an operating system function that is called by the process and the call returns once the asynchronous operation finishes. The calling process (thread) is suspended while the blocking call lasts and does not waste any CPU cycles. A typical example that uses a blocking call may look like this:

```
do_some_work1();
op=initiate_asynchronous_operation();
do_some_work2();//optional
res=block_and_wait(op);
do_some_work3(res);
```

This approach does not work well when used inside a TBB task. The reason is, that TBB maintains a fixed size pool of worker threads. The number of threads is usually the same as the number of threads that can be executed by all of the machine's CPUs at one time, i.e., the number of CPUs multiplied by the hyper-threading (HT) factor, if HT is enabled. If a thread gets blocked inside a task, it cannot be used for any other purpose, since a TBB task cannot be preempted by the scheduler. This leads to sub-optimal CPU utilization. As a result, it is usually recommended to avoid blocking calls [6].

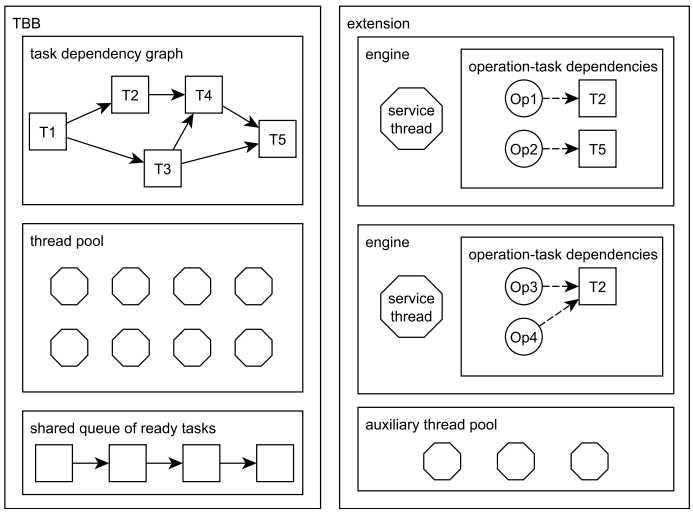
However, we can slightly restructure the original example to make it look like this:

```
task1 {
  do_some_work1();
  op=initiate_asynchronous_operation();
  do_some_work2();//optional
}
task2 { res=block_and_wait(op); }
task3 { do_some_work3(res); }
task1 => task2 => task3 //dependencies
```

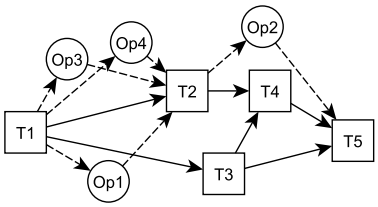
When split into these three tasks, there is a dependency forcing that `task3` can only run after `task2` finishes. Tasks `task1` and `task3` are not a problem for the TBB library, since they do not block. However, `task2` is still a problem that cannot be overcome with the TBB scheduler and the worker thread pool. Our solution was to move this task outside of TBB and make the blocking calls a special entity that is handled differently from the "normal" computational tasks. So, `task2` becomes `operation1` and `task3` now depends on `operation1` rather than `task2`. Only `task1` and `task3` are handled by the TBB, `operation1` is handled by an external entity (we call this entity *engine* in the following text) that does not use TBB worker threads, so they do not get blocked. Instead, a dedicated *service thread* is used by the engine to make the blocking calls.

A potential problem with this solution is demonstrated by Figure 1a. Once `operation1` finishes, `task3` is ready to run. But to run a task, a TBB worker thread must first become available (i.e., not executing a task). However, if all TBB threads are currently busy executing long running tasks, no thread is available to execute `task3`. As a result, a long time may pass between `operation1` finishes and `task3` starts – we call this timespan *latency*. Depending on the application, this may or may not be a problem.

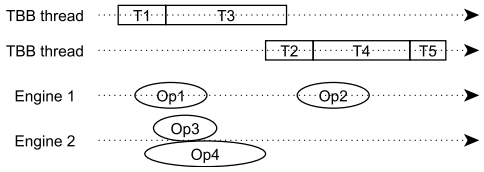
To improve flexibility of our solution, we have implemented different strategies that may be used to start `task3`. These provide different tradeoffs between minimizing the latency and minimizing the total overhead. It is possible to specify a strategy separately for each task. The available strategies are illustrated in Figure 1. The *normal* strategy just submits the task to TBB, the *short task* strategy executes the task within the service thread of the engine, and the *ASAP* strategy uses an auxiliary thread pool to execute the tasks.



(a) TBB and the extension – overview of the architecture



(b) Combined dependency graph. Note that arrows going from tasks to operations represent creation, not dependencies.



(c) One of the possible execution timelines

Fig. 2: Overview of the extension showing a single dependency graph distributed between the individual software components, combined into a single graph, and one of the valid execution timelines that observes all of the dependencies.

III. Application in HyPHI

We have implemented a library called HyPHI [5] which provides TBB-like algorithms that execute hybridly on the host and Intel Xeon Phi coprocessors. Since (by design) the host part of the library is very similar to TBB, we wanted to use the TBB scheduler in the implementation. However, data and execution offloading requires long-running operations that potentially block. The initial versions of our library used TBB for host execution and a set of dedicated threads for offloading. This solution was hard to maintain and extend. Hence, we have developed a TBB extension and different engines. The engines used by HyPHI are: SCI messaging (very similar to network sockets), SCI-provided DMA transfers, and standard network sockets.

For example, to offload a batch of items for processing, the host initiates a DMA transfer to send the data and creates a task that executes once the DMA finishes. This continuation task sends a SCI message to the coprocessor to start processing the data. Another task is created on the host and set up to start after the host receives a message from the coprocessor saying that the data was processed. This task initiates a second DMA transfer to copy the processed data back to the host and creates yet another task that is a successor to the DMA transfer. All communication with the coprocessors is handled like this.

One of the advantages of this approach is that it makes it easy to start multiple tasks in parallel and use dependencies to synchronize them. For example, in HyPHI it is possible that a buffer that is selected to receive offloaded data on the coprocessor is not large enough to contain all of the data. In this case, a message is sent to the coprocessor to increase the size of the buffer, but at the same time a new task is created on the host to prepare (serialize) the data for offloading. The DMA operation that sends the data to the coprocessor is started in a task that can start only after both the serialization and buffer size increase are done. So, this task is a successor of the serialization task and the "buffer expanded" response from the coprocessor.

IV. Architecture overview

The overall architecture of TBB and our extension is shown in Figure 2. The extension is completely outside of TBB, so the user can use the TBB library without any modification and optionally also include our extension. The extension consists of two parts: the individual engines that deal with blocking operations and an optional auxiliary worker thread pool. Each engine should deal with all blocking operations that can be served by a single service thread. For example, one engine can deal with all operations that provide a file descriptor that can be supplied to a `select` blocking call, like network sockets. A different engine (and associated service thread) can deal with SCI communication channels (used by the Intel Xeon Phi Coprocessor) or with OpenCL calls. Each engine also tracks all dependencies that have a blocking operation handled by the engine as the predecessor. As shown in Figure 2, a single task (T2) may be a successor in dependencies maintained by multiple engines, but also in the original TBB dependency graph. This is made possible by a common software component (supplementary task scheduler) that is used by all engines when a task's predecessor (blocking) operation finishes. This piece of code determines whether the task is ready to be executed and if so, makes sure that it gets executed at some point in time, which is not necessarily immediate.

A. TBB scheduler and task graph

The TBB scheduler maintains a fixed size thread pool. In fact, each thread has a local scheduler and these together play the role of a global scheduler, but very little shared data is actually used in order to avoid bottlenecks and scalability issues. Usually, the number of threads is equivalent to the number

of logical cores (i.e., number of physical cores multiplied by the hyper-threading degree). It is possible to set the number of threads manually when the scheduler is initialized to any value. The number of threads may even be larger than the number of logical cores. This could be used to somewhat reduce the negative impact of blocking calls, but obviously only to some degree and if the number of such concurrent calls is small. Each local (per-thread) scheduler maintains a queue of tasks to be executed. There is also a global (shared) queue of tasks to be executed, which is served by all local schedulers. [1]

There exist several groups of tasks. First, there are tasks that have dependencies that have not been fulfilled. Such tasks are *waiting* or *not ready*. The second group are tasks without pending dependencies but they are not yet executing. These tasks are *spawned* (in local scheduler's queue) or *enqueued* (in global queue). The third group are currently *running* tasks. A task is thread-bound (if it starts executing on a thread, it executes only on that thread) and non-preemptive. Tasks can only be separated from a thread (stop executing) by an explicit call to a TBB-provided function that the thread makes.

The task dependency graph is implemented by assigning one pointer and one (atomic) counter to each task. The pointer points to a *successor* – a task that depends on the current task. The counter is in fact a reference counter. It reflects the number of *predecessors* – tasks that have the current task as their successor. The pointer can be a null pointer, meaning that no other task depends on this task. An important feature of the reference counter is that it can be (and often needs to be) adjusted by the user. It is not required for the counter to be equivalent to the actual number of predecessors. For example, setting the counter to be one more than the number of predecessors ($N+1$) means that the task cannot start even if all predecessors finish. To allow the task to start, the reference count must be decremented by calling the appropriate method.

When a task finishes, the scheduler automatically decrements the counter of its successor. If the counter reaches zero, the successor is transferred to the queue of tasks to execute – the task is spawned. If some code outside of TBB decrements the counter, for example to reflect the fact that a task with $N + 1$ reference count can start executing as soon as all predecessors have finished, it must check whether the counter has reached zero and if so, spawn the task.

B. Task graph extension

The way TBB tracks task dependencies provides a large degree of freedom to the library user. This gave us an idea that we could incorporate blocking operations into TBB, but not by allowing the user to make blocking calls. We decided to add a new node and edge type into the task graph. So far, only task-to-task (predecessor-to-successor) edges have been supported. We want to provide another type of dependency, when a task depends on a blocking operation. So, the blocking operation becomes a new node in the graph and it is linked to a task that should not run before the blocking operation finishes.

C. Implementation

As we have already mentioned, the blocking call is not made as part of a task's execution. The task just initiates an asynchronous operation. It is not the task's responsibility to wait for the operation to finish. That would probably require the task to block, which we want to avoid. But *someone* has to do the waiting. This means that there must be a thread, since a thread is required for anything to run. The thread cannot be part of the TBB thread pool, since those threads should not be blocked. So, as a result, we created an extra thread, called service thread, to invoke the blocking operation.

This service thread can block. In fact, it should be suspended (blocked) most of the time, to minimize the amount of CPU cycles it "steals" from the TBB threads. If the service thread is woken up, it means that a blocking operation has finished. The service thread should quickly handle this situation and suspend again. In our setup, the service thread needs to decrement the reference counter of the successor of the finished operation and spawn the successor if its reference count reached zero. Fortunately, these are very simple and fast operations.

Ideally, one service thread should be enough to service multiple concurrent asynchronous operations. This should be possible since the operating system usually provides a call that can be used to specify multiple operations and wait for any of them to finish, e.g., `select` or `poll` on Linux or an I/O completion port on Windows [7]. The amount of work that has to be done by the service thread upon completion of an operation is very small. One service thread is easily capable of spawning tasks for many TBB worker threads. This is because an average TBB task is expected to do non-trivial amount of work to keep the total overhead small.

D. Task spawning

In the previous description, we said that the task has to be spawned by the service thread, if the reference count reaches zero. However, spawning a task in TBB means adding it to the queue of ready tasks maintained by the task scheduler assigned to the current thread. Since the service thread is not a TBB thread, there is no scheduler and no task queue. An easy way to get around this problem is to enqueue the task. When enqueued, the task is added to the shared task queue, so no local scheduler is needed. The task will then eventually be executed by a TBB thread. The shared queue is being served by all worker threads, if they have no work of their own. In such case, the scheduler of such thread first looks for a task in the shared queue and if no task is present, it steals a task from another thread.

A scheduler looks at the shared queue if its local queue is empty. However, if the thread is executing a long running task, the scheduler is not active at all until the task finishes. So, if all TBB worker threads are busy with long running tasks, if their local queues are full of tasks, or if the running tasks keep spawning new tasks (creating new tasks is not enough, they have to be marked "ready to run" as well) then the shared queue may not be served for a long time. In that case, we may get a large latency.

E. Code examples

The following code examples demonstrate an example engine. SCI messages are used as the example. The following code shows a way in which a task issues a command to the Xeon Phi coprocessor and creates a task that processes the response. Note that constructors and other non-essential code have been omitted for brevity.

```
struct initiator : public tbb::task
{
  task* execute()
  {
    //create a task to handle the response
    task* t=new(allocate_continuation)processor(s);
    //set t's number of predecessors to 1
    t->set_ref_count(1);
    //create dependency from s to t
    engine::register_successor(s,t);
    //issue the command
    send_command_over_sci(s);
    //note that t can start executing at any point
    //after the command is sent, so the preferred
    //way is to first set up all dependencies and
    //then start the blocking operation
    return 0;
  }
  sci_endpoint s;
};
struct processor : public tbb::task
{
  task* execute()
  {
    //we know that data is ready on the endpoint s
    process_response(s);
  }
  sci_endpoint s;
};
```

V. Latency reduction

The main sources of latency are tasks that have been enqueued by a service thread. If all TBB threads are occupied, the task may spend a long time in the shared queue, waiting for a thread to run out of tasks, because only then the thread checks the shared queue for work to do.

To better understand different strategies for reduction of latency, we describe the following situation: the service thread is currently running – it is outside the blocking call. There is a task T_{run} ready to be executed. All TBB threads are busy.

a) Normal strategy: Even though it may result in a long latency, enqueuing the task is still a viable option. The advantage of this solution is that it provides the best environment for the TBB scheduler to work as efficiently as possible.

b) Service thread strategy: One option is to have the service thread execute T_{run}. This has the advantage of getting T_{run} to run as soon as possible. But there are several issues. First, while T_{run} is running, the service thread is unable to check whether other operations have finished. Second, this solution also does not scale, since there is just one service thread. Third, T_{run} may specify (return) another task to be executed, the successor's reference count may reach 0, or it may spawn other tasks. These tasks also have to be executed which then results in a similar situation as with the original T_{run}, but possibly with more tasks to deal with. These issues make the approach only suitable if T_{run} is very short and if it does not spawn any additional tasks.

c) Auxiliary thread pool strategy: Another option is to create extra threads specifically to handle task execution. This supplementary thread pool can be used to run T_{run} plus additional successor tasks. If no tasks are present for execution, the threads are suspended. This means we may have more running threads than (logical) cores in the system. This oversubscription may not be ideal, but it allows us to trade some of the efficiency for reduction of latency. This solution does not prevent the service thread from doing it's core work and it is also able to scale better, since the thread pool may contain more than just one thread. On the other hand, if T_{run} (or the other tasks it spawns) takes very long to finish, the system would still get congested and the latency will increase. So, this approach is suitable if the work done by T_{run} and its successors is small compared to the work being done by tasks executed on the normal TBB threads.

d) ASAP strategy: The previous three strategies can be combined to create new strategies. For example, it is possible to combine TBB and non-TBB thread pools like this: instead of enqueuing T_{run}, two new tasks T_{proxy} are created. Both proxy tasks contain a pointer to T_{run}. First of them is enqueued with TBB, the other one is submitted for execution by the non-TBB thread pool. Using atomic operations, we make sure that (only) the first copy of T_{proxy} that gets started runs the T_{run} task. This way, we maximize the chances of running T_{run} as soon as possible, but the added overhead is still small and also constant for any task.

e) Short task strategy: Another option is to execute the T_{run} on the service thread and handle all new tasks spawned by T_{run} according to the ASAP strategy.

When a operation-task dependency is added, the call has an argument where the user specifies the strategy. The service thread and auxiliary thread pool strategies are not provided by our implementation, since in our experience the more advanced alternatives (short task and ASAP) have proven to be more suitable.

VI. Experiments

We have executed a set of experiments to evaluate the performance of our TBB extension. To minimize noise, we decided not to use disk or network operations. Similarly, the overhead created by blocking operations (with the TBB add-on) in HyPHI is much lower than effects caused by other factors. Hence, for better assessment of our extension, we created a special engine that issues a blocking `sleep` operation. The engine allows the developer to specify that a task can only start after a certain amount of time has elapsed. The experiment executed two different types of chains of tasks (a sequence of tasks that have predecessor-successor relationships). The first type consists of 8 tasks, each performing a CPU-intensive computation. No blocking operations are used. The other type consists of 4 tasks, each executing only 1% of the work (compared to the first type), but a successor in the chain

TABLE I: Performance of different solutions and strategies

variant	time (s)	latency (ms)
TBB – no blocking operations	6.71	N/A
TBB – blocking calls in TBB tasks	9.71	N/A
TBB ext. – ASAP strategy	6.96	5.53
TBB ext. – short task strategy	7.13	0.00
TBB ext. – normal enqueue strategy	6.94	426.49

can only start running 1 second after its predecessor started running. As a result, the second type of task chain cannot finish in less than 4 seconds. On the other hand, the tasks do such a small amount of work, that it is possible to finish the whole chain in little over 4 seconds, while the first type of task chain takes well over 6 seconds. We have used 32 chains of the first type and 4 chains of the second type, to simulate an application that does some heavy computation and a bit of communication or synchronization. We consider this to be the main use-case for our TBB extension. The experiments were performed on a server with two Intel Xeon E5-2650 CPUs, featuring a total of 16 physical CPU cores and 32 hardware threads (hyperthreading was enabled on the machine). The TBB thread pool consisted of 32 threads. There was one service thread for the engine and the auxiliary thread pool contained one additional thread. We executed each experiment 14 times in a row. The first four runs were used to "warm up" the runtime. The results from the next ten runs were averaged to produce the final numbers.

Table I shows the performance of the different approaches. It shows the average run-time of the ten runs and average latency of all operation-task dependencies from these ten runs. The first row is provided for comparison and shows the performance when all blocking operations are removed. The second row demonstrates that invoking blocking operations directly from a TBB task significantly reduces the performance. The last three rows show the performance achieved by our TBB extensions when different task spawning strategies are used. As observable in Table I, using `enqueue` and the TBB thread pool achieved the best performance. Launching the task directly in the service thread (short task strategy) eliminates latency but also reduces performance. The ASAP strategy is a tradeoff between performance and latency.

It is important to note, that the concrete results depend on the experimental setup. With an experiment setup like we did, the normal enqueue strategy provides the best performance. However, the experiment was designed in such a way that the long latency did not affect the overall performance. If we carefully increased the sleep interval, we would eventually reach a point where the latency starts to play a major role and the ASAP strategy would outperform the normal strategy. On the other hand, if the tasks that follow the blocking operation are trivial, the short task strategy can outperform the others.

VII. Related work

There are multiple libraries that provide asynchronous I/O operations. However, they were mostly designed to be used in conjunction with process-based or thread-based parallelism. Operating system support or Boost.Asio [8] fall into this category. These libraries would typically be used to build an engine in our framework. They do not deal with task dependencies and task spawning.

The Microsoft .NET framework now supports the task-based Asynchronous Pattern (TAP) [9], which is much closer to our problem area. This solution is similar to ours since starting an asynchronous operation returns a `Task` object that can then be used to create dependencies. E.g., tasks can be set to run after the asynchronous operation has finished. Our extension does not provide a task object since tasks and dependencies in TBB are created and managed in a different way than in .NET. In addition, explicit engines as in our approach are not used by the .NET solution. This is possible thanks to the sophisticated support that .NET provides for asynchronous operations (the `async` and `await` keywords) in the compiler, runtime and I/O libraries. For example, if the asynchronous operation provides a callback, the callback can be easily transformed into a task that executes when the operation finishes.

VIII. Conclusion

We have designed and implemented a TBB extension that allows asynchronous operations to be included in TBB task dependencies. This way, it is much easier to implement applications that use TBB extensively to handle computation, but also need to perform I/O operations. Our experiments show that there is some added overhead, but it is reasonable. Compared to blocking calls, however, our extension can achieve significant performance improvements. The extension has been successfully used by the HyPHI library, which automates fully hybrid execution of parallel patterns on systems with Xeon Phi coprocessors.

Acknowledgment

This work was partially supported by the European Commission's FP7, grant no. 288038, AutoTune.

References

[1] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in Intel Threading Building Blocks," *Intel Technology Journal*, vol. 11, no. 04, pp. 309–322, November 2007.

[2] R. Farber, *CUDA Application Design and Development*, ser. Applications of GPU computing series. Morgan Kaufmann, 2011.

[3] A. Munshi *et al.*, "The OpenCL Specification version 1.2," 2012, khronos OpenCL Working Group, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (last visited July 2013).

[4] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Elsevier Science & Technology Books, 2013. [Online]. Available: http://www.sciencedirect.com/science/book/9780124104143

[5] J. Dokulil, E. Bajrovic, S. Benkner, M. Sandrieser, and B. Bachmayer, "HyPHI – task based hybrid execution C++ library for the Intel Xeon Phi coprocessor (to appear)," in *42nd International Conference on Parallel Processing (ICPP-2013), Lyon, France*, 2013.

[6] "Threading Building Blocks Documentation." [Online]. Available: http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/

[7] "I/O completion ports." [Online]. Available: msdn.microsoft.com/en-us/library/aa365198.aspx

[8] C. Kohlhoff, "Boost.Asio." [Online]. Available: http://www.boost.org/doc/libs/1_54_0/doc/html/boost_asio.html

[9] "Task-based asynchronous pattern (TAP)." [Online]. Available: http://msdn.microsoft.com/en-us/library/hh873175.aspx