HyPHI – task based hybrid execution C++ library for the Intel Xeon Phi coprocessor

Jiri Dokulil, Enes Bajrovic, Siegfried Benkner, Martin Sandrieser
Research Group Scientific Computing
University of Vienna
Austria
Email: firstname.lastname@univie.ac.at

Beverly Bachmayer
Software and Solutions Group
Intel GmbH
Germany
Email: bev.bachmayer@intel.com

***Abstract*—The Intel Threading Building Blocks (TBB) C++ library introduced task parallelism to a wide audience of application developers. The library is easy to use and powerful, but it is limited to shared-memory machines. In this paper we present HyPHI, a novel library for the Intel Xeon Phi coprocessor for building applications which execute using a hybrid parallel model that exploits parallelism across host CPUs and Xeon Phi coprocessors simultaneously. Our library currently provides hybrid for-each and map-reduce. It hides the details of parallelization, work distribution and computation offloading from users while using internally TBB as its foundation. Despite the higher level of abstraction provided by our library we show that for certain types of applications we outperform codes that rely on the built-in offload support currently provided by the Intel compiler. We have performed a set of experiments with the library and created guidelines that help the developers decide in which situations they should use the HyPHI library.**

I. Introduction

The novel architecture of the Intel Xeon Phi coprocessor enables new ways of approaching accelerated application development. Even though it is not 100% binary compatible with current x86-64-based processors, (the vector unit is different from the SSEx available in the today's CPUs), it is usually possible to use the same principles to develop applications for the coprocessor that one would use to develop a traditional parallel application. The main difference is the fact, that thread-level parallelism and vectorization play a more important role, due to the higher number of cores and greater emphasis on vector units.

As a result of the fairly conventional architecture, well-designed parallel libraries targeting traditional shared-memory systems may be used efficiently on Intel Xeon Phi coprocessors. An example of such a library is the Intel Threading Building Blocks (TBB) [1]. This parallel C++ library introduced the idea of task parallelism to a wide audience of developers. It provides a set of parallel algorithms, for example `parallel_for`, `parallel_reduce`, and `parallel_pipeline`. These algorithms offer a high-level programming approach for common computational patterns but fully hybrid execution, exploiting all of the host- and coprocessor-cores at the same time, is not supported. The Intel C++ Composer XE 2013 compiler already offers means for offloading computational tasks from the host to coprocessors. This can greatly simplify development of *accelerated* applications that utilize Xeon Phi coprocessors – a large fraction of the work required for offloading is done automatically by the compiler and runtime-system. However, with this support it is difficult to achieve hybrid execution and dynamically distribute the work to the host and multiple coprocessors.

In this paper we present HyPHI, a high-level parallel library for Xeon Phi that we designed to support fully hybrid execution and dynamic work distribution, while maintaining the design principles and ease of use of TBB.

The contributions of this paper are the following:

- We show the usage of existing programming approaches for Xeon Phi and give examples how the HyPHI library can be utilized for fully hybrid execution.
- We introduce the design principles of HyPHI based on well-established C++ programming techniques and provide details about the implementation such as dynamic work distribution, data offloading and task scheduling.
- We have implemented dynamic work distribution, hybrid *foreach* and *map_reduce* constructs, data serialization facilities as well as support for different work offload strategies for performance tuning.
- We evaluate our library via a comprehensive set of benchmarks as well as a real-world physical simulation application. In addition, we provide guidelines that advise programmers when to use our library.

The remainder of this paper is organized as follows: Section II briefly introduces the Xeon Phi hardware and key components of the software stack. Section III discusses the support provided by the Intel compiler for offloading computations to the Xeon Phi and outlines the parallel algorithms provided by our HyPHI library. Section IV provides details about the design and implementation of the library. Section V presents experiments with synthetic benchmarks that aim at deriving guidelines for using our library, and presents performance results for a real world simulation application. Related work is discussed in Section VI followed by a summary and conclusion in Section VII.

II. Xeon Phi Ecosystem

Xeon Phi coprocessors comprise more than 50 Intel Architecture (IA) cores, each with four way HyperThreading, to produce the total of over 200 logical cores. Additionally, Xeon Phi coprocessors include memory controllers that support the GDDR5 specification and special function devices such as the PCI Express interface. Xeon Phi cores run independently of each other and have very powerful vector units. The memory subsystem provides full cache coherence. Cores and other components of Xeon Phi are connected via a ring interconnect. From the software point of view, a Xeon Phi coprocessor is a shared-memory computing domain, which is loosely-coupled to the computing domain of the host.

The Xeon Phi coprocessor is implemented as a PCI Express form-factor add-in card. The high-level software architecture of a system with a host and Xeon Phi coprocessor is depicted in Figure 1. The host software stack is based on a standard Linux kernel. The software stack for the Xeon Phi is based on a modified Linux kernel. The operating system on the Xeon Phi coprocessor is in fact an embedded Linux environment that provides basic functionality such as process creation, scheduling, or memory management.

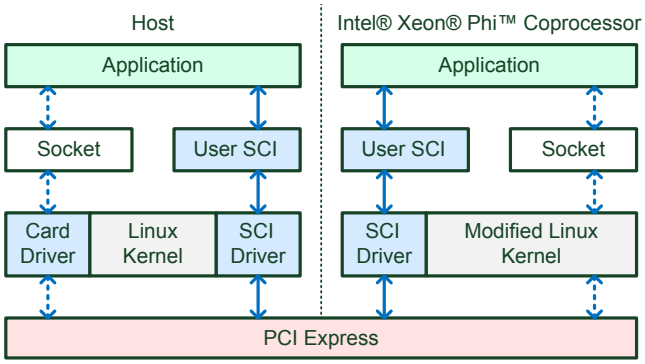

Figure 1. Software architecture of a system with a host and a Xeon Phi coprocessor. Acronym: Symmetric Communication Interface (SCI).

Multiple options are available for communication between the host and the card. The card driver provides virtual network interfaces, so it is possible to use the TCP/IP network stack. This is good for management and compatibility with existing applications. On the other hand, it cannot provide maximum performance, since the network stack was designed for a different purpose than communication over PCI Express.

The specialized SCI (Symmetric Communication Interface) library provides two communication options. CPU based messaging interface similar to sockets and DMA transfers. It is a low-level library so there is minimal overhead. Its usage is similar to existing communication libraries for sockets and DMA transfers.

The Intel C++ Composer XE compiler is used to compile applications targeting the coprocessor. This can be done either by cross-compiling the sources for the coprocessor, or by using the compiler's support to build and easily deploy offloaded applications. These applications are started on the host, but a binary image of the code compiled for the coprocessor is automatically transferred to the card, started within a process on the cards, and connected to the host's process by SCI. The following section explores this option in further detail.

III. Comparison to existing approaches

The Intel C++ Composer XE 2013 compiler provides two ways of building offloaded applications. Both approaches enable the programmer to add Xeon Phi support by specific source-code annotations. The first uses C/C++ `pragmas` to mark code regions for execution on the coprocessor. Input and output data for each region has to be specified explicitly. The second approach is based on implicit data-management by using special memory allocation routines for *shared* data in conjunction with specific *offload* keywords. Both approaches do not support automatic exploitation of all available host-cores and coprocessors in a system.

This is where the HyPHI library comes in. Unlike the general-purpose offload solutions provided by the Intel compiler, we provide a set of higher-level *algorithms* implementing computational patterns such as *for-each* with fully hybrid execution support and dynamic work distribution among all available processing resources (host-cores and coprocessor cores).

In the following text, we present basic usage examples of the *traditional* offload approaches and show how our library can simplify hybrid application development. In all of the examples, the goal is to apply a functor (function object) to an array or vector with data. The functor looks like this:

```
struct functor {
  int factor;
  functor(int f) { factor=f; }
  void operator()(int& x) const {
    x=x*factor;
  }
};
```

A. Using pragma-based offload

For the *pragma*-based explicit offloading support the example may look like this:

```
int *data=new int[size];
fill_data(data);
#pragma offload target(mic) inout(data:length(size))
{
  functor f(2);
  tbb::parallel_for_each(data,data+size,f);
}
```

The block of code after the `#pragma offload` is executed on the coprocessor, the array `data` of the size `size` is sent to the coprocessor before the block starts executing and is sent back to the host, after the block

finishes. The execution on the host resumes once the data is back on the host. Only "plain" data structures (bitwise copyable, without pointers, single value or contiguous array of values) can be used.

B. Using shared-memory offload

The second option is using *offload* with shared-memory. Since the Xeon Phi coprocessor uses 64bit memory address space (just like the host), it is possible to allocate memory in the host's address space and the coprocessor's address space with the same address. This enables using pointers inside of data structures, even with extensive use of dynamic memory allocation. Global variables are also supported – the variable may be marked as shared and then the value is synchronized between the host and the coprocessor. This is not possible in the `pragma`-based approach. It is important to note, that the shared memory offloading does not provide a true distributed shared memory. The consistency is only maintained at two points: when the offloaded block starts and when the offloaded block ends.

An example code may look like this (using OpenMP to parallelize execution on the coprocessor):

```
_Cilk_shared void process_data(int* data, int size)
{
  functor f(2);
  #pragma omp parallel for
  for(int i=0;i<size;++i) {
    f(data[i]);
  }
}
...
int* data=_Offload_shared_malloc(sizeof(int)*size);
fill_data(data);
_Cilk_offload process_data(data,size);
```

C. Hybrid Execution via existing approaches

Both offloading approaches lack out-of-the-box support for hybrid execution. In addition, offloaded sections block execution of the main host process. Using multiple coprocessors and the host-cores simultaneously therefore requires manual parallelization for all computational tasks. Since the host system and each coprocessor have distinct memory regions, manual parallelization is challenging. Shared-memory can only be synchronized when offload execution starts or finishes but not during task execution.

However, it is possible to statically split the work and use conventional threading for manual parallelization. The following example demonstrates a simple case where two coprocessors are used with mutiple host-threads and offload `pragmas`:

```
int *data=new int[size];
fill_data(data);
int *data2=data+(size/2);
std::thread t1([&](){
#pragma offload target(mic:0) inout(data:length(size/2))
{
  functor f(2);
  tbb::parallel_for_each(data,data+size/2,f);
}
});
std::thread t2([&](){
#pragma offload target(mic:1) inout(data2:length(size/2))
{
  functor f(2);
  tbb::parallel_for_each(data2,data2+size/2,f);
}
});
// Do some work here or run t2's work in the main thread
t1.join();
t2.join();
```

As shown in the previous listing, programmers have to manually split data into parts and concurrently issue commands to process each part on a coprocessor. However, it may not always be possible to determine the optimal size of chunks. This issue gets aggravated when processors with different performance characteristics (such as host and coprocessors) participate together in the computation.

D. Using the HyPHI library

In contrast to the previously introduced approaches, the HyPHI library provides fully hybrid execution and dynamic work distribution. This aims at simplification of programming support when all available system resources need to be exploited.

In the case of the first example in this section, the code would look like this:

```
std::vector<int> data;
fill_data(data);
functor f(2);
hybrid_for_each(data.begin(),data.end(),f);
```

Compared to the established *for-each* functionalities provided by STL (C++ Standard Template Library) or TBB, only one additional aspect needs to be considered for our library. Due to the distributed memory regions and required transfers to/from coprocessors, functor data and items must be serialized/deserialized. For complex data-types, HyPHI provides functionality for specifying custom (de-)serialization routines. Basic data-types (such as the `int` items in the example listing) and trivially copyable structures (can be copied byte by byte, e.g., the functor's data in the example) are (de-)serialized automatically by the library.

The `hybrid_for_each` algorithm is in fact a C++ template function. It works with an input specified by two forward iterators: beginning and end of the data. The action is specified by the third parameter – the functor. Data may be in a plain C array, C++ standard template library container (e.g.,`std::vector`) or any other structure that provides these iterators. The functor is a function object with similar requirements as those imposed by TBB. These are a superset of requirements of the C++ standard. However, for offloading, the function object data must also be serializable. Furthermore, it must be marked (using `pragmas`) as offloaded code, so that the compiler includes it in the coprocessor binary. This is true for all source codes that are executed on the coprocessor, e.g., helper functions used by the functor. The relevant parts of the library are also marked this way.

Thanks to the familiar interface of the hybrid algorithms, the application code can be easily rewritten from hybrid to parallel (using TBB) or serial (using the standard C++ library) just by changing the call from `hybrid_for_each` to a different function:

```
tbb::parallel_for_each(data.begin(),data.end(),f);//TBB
std::for_each(data.begin(),data.end(),f);//serial
```

The HyPHI library also provides `hybrid_reduce`, which is a hybrid alternative of the `parallel_reduce` algorithm provided by TBB. It accepts input data the same way as `hybrid_for_each`, but it needs two functors. The *map* functor, which is applied to every item in the sequence to produce a new value, and *reduce* functor that is used to combine these values (and intermediate results) to produce a single value.

The hybrid reduce may be used like this to multiply each item by two, convert it to integer, and sum all the values:

```
struct map {
  int operator()(double x) {
    return (int)(x*2);
  }
};
struct reduce {
  int operator()(int l, int r) {
    return l+r;
  }
};
int go(std::vector<double>& data) {
  return hybrid_reduce(data.begin,data.end(),
    0,map(),reduce(),2);
}
```

The zero in the argument list of reduce denotes an identity value for the reduce operation. Via the optional last integer argument of reduce and for-each, the number of coprocessors to use can be given (2 in the example above). This argument can also be 0, in case host-only execution is desired (i.e., for debugging). If the argument is left out, all available coprocessors are used.

IV. Design of the Library

We have designed the HyPHI library to provide developers with a simple way to integrate support for Xeon Phi coprocessors in their C++ applications. HyPHI transparently supports hybrid execution and dynamic work distribution with a familiar C++ interface. The library may be combined with other parallel programming approaches, e.g., TBB may be used in parts of the application or even in user-provided functors passed to the library.

In the following text, we will describe the overall architecture of the library and also provide a more detailed description of the most important components of the library.

A. Overall architecture

The HyPHI library follows the design of the TBB library. At the core of TBB is a task scheduler. This scheduler maintains a thread pool consisting of a fixed number of general purpose worker threads. The threads from this pool are then used internally by the scheduler to execute tasks created during the execution of the parallel algorithms, while observing task dependencies specified by the implementor of the algorithms [1]. With this approach, there is no direct mapping between tasks and execution threads, so it is possible to create any number of tasks without overprovisioning the system.

The HyPHI library is built on the same principles and uses the task scheduler from the TBB library as its foundation. However, for supporting long running operations such as data transfers, we had to extend the available task dependency support to also consider SCI messages and DMA transfers. The implementation is an add-on for the TBB library which is completely outside the TBB source codes.

Our `hybrid_for_each` and `hybrid_reduce` algorithms are realized as C++ template functions that execute user-provided functor(s) on the user's data in a pre-defined manner. These algorithms are implemented by a set of tasks that are combined (using task dependencies) to perform the desired action.

A high-level view of the way in which the HyPHI library executes the `hybrid_for_each` algorithm is shown in Figure 2. As you can see from the figure, the items are processed in batches. The host's part of item processing (on the very left) is performed by a few management tasks that split (in parallel) the batch to individual items and then create execution tasks that do the actual work, i.e., apply the functor to the item. The number of items in a batch (and also the number of execution tasks) is equal to the number of host's cores. The processing of subsequent buffer overlaps with processing of the current buffer, to eliminate adverse effects caused by the fact that some items may take longer to process than the others.

The box in the middle takes care of computation offloading. It has two parts. The first serializes items and sends them to the coprocessor for processing, the second receives serialized representation of processed items, deserializes it, and stores the results in the array. The data batch is always processed as a whole during these offloads, so no splitting is performed. The management tasks use SCI messages to synchronize their work with the coprocessor and DMA (also provided by SCI) to transfer the data.

Local processing and offloading to the coprocessor is carefully coordinated by special synchronization tasks (barriers) and some shared state. These tasks also control the number of buffers being used on the host and the coprocessor.

The coprocessor process is created using offload `pragmas`, but `pragmas` are only used to copy the binary and launch the process, not to transfer data and control execution. These tasks are handled by SCI messages and DMA.

Item processing on the coprocessor is similar to the item processing on the host, but there are several significant differences. First, the number of execution tasks is much larger to reflect the much larger number of execution threads

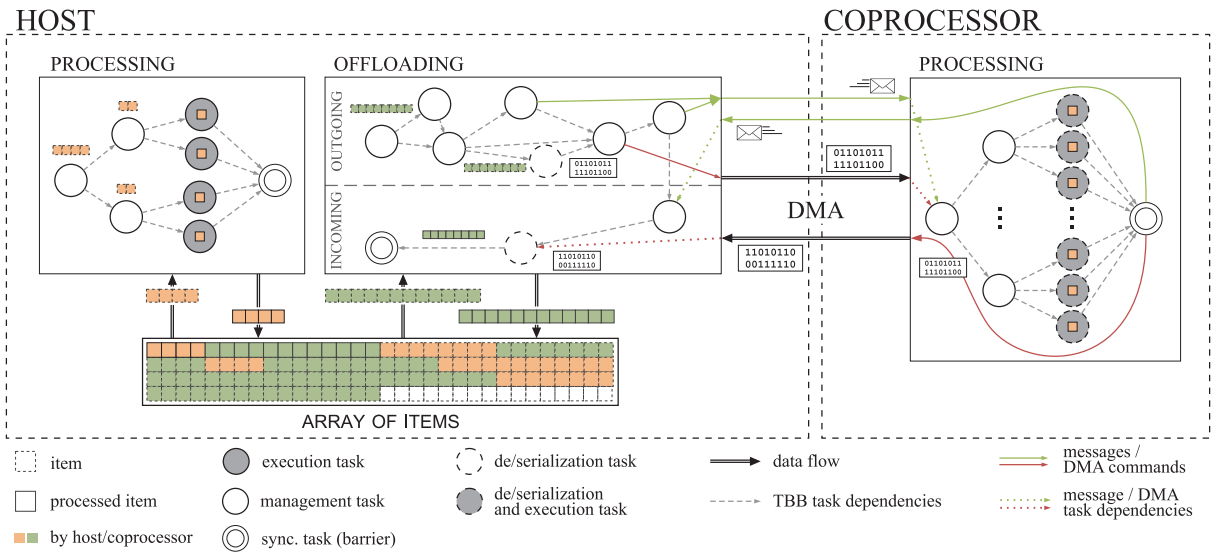

Figure 2. High-level overview of execution of the hybrid for-each algorithm.

on the coprocessor. Second, the execution tasks also deal with deserialization and serialization of the items. Third, execution is coordinated using commands (messages) from the host, since the shared state is not available on the coprocessor.

B. Work distribution

Our hybrid algorithms distribute the work between the host and the coprocessors dynamically. Both the host and the coprocessor request work in larger chunks (tens of items for the host, hundreds of items for the coprocessor) that are then processed in parallel. Using larger chunks on the host and the coprocessor may reduce overhead and improve performance. However, finding the right granularity might be very application specific. To deal with this, the library supports different tuning strategies (see also Section IV-E).

On the host, a chunk is requested when it is needed, i.e., when a worker thread on the host is idle. For the offloaded items, the library tries to maintain (at least) double buffering, to minimize the amount of time that the coprocessor spends waiting for work to process. This means that while a batch is being processed on the coprocessor, the next batch is being prepared on the host and transferred to the coprocessor.

C. Task dependency extension

The HyPHI library uses the TBB library both as inspiration and foundation. Just like in the TBB, the algorithms that HyPHI provides are built using tasks and task dependencies. However, since blocking tasks should be avoided, we use one seperate thread to start the code running on the coprocessor (via Intel's offload). This avoids stalling a thread from the TBB thread pool.

The task-based approach to parallel programming can be very flexible. To deal with synchronization, usually task dependencies are used. A dependency between task A (the predecessor) and task B (the successor) means that the task B cannot run before the task A finishes [1]. The implementation of the whole mechanism is simple and can be efficient. However, building a library that efficiently deals with data offloading is challenging. Communication such as messages and DMA transfers may require blocking calls which could lead to inefficiencies with the task-based approach.

Fortunately, in our case there are just two types of blocking calls that we need to make – connection polling and DMA wait-for-completion. So, we have extended the TBB library to support additional types of dependencies than the task-task dependency. With our extension, a task A may be registered as a successor to a connection C. In this case, task A cannot run until there is a message available on the connection C. The second type of dependency is a DMA-task dependency. This means that the dependent task cannot run until the specified DMA transfer finishes. There may be any number of task-task, connection-task and DMA-task dependencies.

The implementation of our extension uses operating system threads to perform the blocking call. This means that for most of the time, the only active (non-suspended) threads in the whole application are TBB worker threads.

D. Offloading items to the coprocessor

At some point during the execution of the hybrid for-each algorithm, it is necessary (the implementation of the algorithm decides this) to offload items $I = (i_1, ..., i_n)$ to a coprocessor. This is achieved by spawning the task $T_{offload}(I)$. The task $T_{offload}(I)$ spawns two more tasks: T_{get_buffer} and $T_{serialize}$. It also creates task T_{send} and sets

it as a successor (creates task-task dependencies) of both T_{get_buffer} and $T_{serialize}$. The task T_{get_buffer} sends a message to the coprocessor, requesting a new buffer for data. Then, it creates a connection-task dependency and sets T_{send} as a successor of the connection. The coprocessor will use this connection to signal that the buffer is ready. As a result, the task T_{send} will only run once the data has been serialized and a buffer on the coprocessor is ready. When that happens, the task will initialize DMA to transfer the serialized data (items I) to the coprocessor. It also creates task T_{notify} and sets it as a successor of the DMA – it creates a DMA-task dependency. Once the task T_{notify} executes, we know that the data has been transfered to the coprocessor. A message can be sent to the coprocessor to invoke the processing of the buffer.

E. Strategies

The usual challenge created by bulk (i.e., using large batches) dynamic work distribution is the phase of the computation close to the end. If buffers are too large or there are too many of them, the host may run out of items to process, while there is still a lot to be done on the coprocessor. On the other hand, if the buffers are too small, the number of items ready to be processed by the coprocessor gets too small and the coprocessor is not fully utilized. To minimize this impact, we dynamically decrease the size of the buffers near the end. However, finding the optimal ratio is not easy, since we have no guarantee about the time it takes to process different items.

For scenarios that feature unbalanced computations, the HyPHI hybrid algorithms may be parametrized by different *strategies*. This concept is analogous to partitioners in TBB. A strategy influences the run-time and dictates the number and size of the buffers it should use. We have implemented three strategies:

Dynamic buffer size: The size of the buffers decreases as the execution nears the end of the sequence. This strategy tries to emphasize host utilization and may be suitable if the workload performs good on the host.

Fixed buffer size: The size of the buffers is constant. As a result, the coprocessor utilization is increased, but the host's resources may not be fully utilized near the end. This is useful if the host has other work to do or the workload is much better suited for the coprocessors.

Pure offload: This strategy completely disables item processing on the host. This is useful if the workload is extremely well suited for the coprocessor or the host has other important work to do.

V. Experiments

We evaluated the HyPHI library via synthetic benchmarks as well as a full application. Configurable synthetic benchmarks allow us to test many different scenarios and thus discover under which condition it is beneficial to use the HyPHI library. In addition, a demanding physics simulation application (smoothed particle hydrodynamics) was evaluated to show the usefulness of our approach.

Experiments have been performed on a machine with two six-core Intel Xeon CPUs (X5680, 12MB Cache, 3.33 GHz, Hyper-Threading) and two Xeon Phi coprocessors (pre-production models – 61 cores, 1.09GHz, 8GB RAM). The source codes were compiled with the Intel C++ Composer XE compiler set to O3 optimization level. All reported times are wall clock times, measured by the internal clock of the host machine. Measured time always includes all of the data transfers. The presented numbers are averages from five consecutive runs of the same experiment.

A. Hybrid for-each loop

In these tests, a function (a long sequence of simple arithmetic operations) was applied to every item in an array (C++ `vector`). The configuration options of this test are the following:

- number of items in container (COUNT) – there is some fixed overhead associated with each call to `hybrid_for_each` and it may amortize better for larger arrays
- size of single item (SIZE) – the relation between size (in byte) of the item and the time it takes to process the item may be an important factor when offloading is concerned
- processing time per item (HOST-TIME) – the average time (in seconds) a single host thread needs to process an item
- processor suitability (PHI-RELATIVE-TIME) – some operations execute better on the host, some on the coprocessor. This may significantly influence the performance gained by offloading. Is the execution time per item on Xeon Phi compared to execution time per item on host.
- processing time balance (BALANCE) – the processing time per item may not be constant, which may influence the efficiency of static work distribution. x means that the second half of items takes x-times longer to process in total than the first half. In other words, if $x = 1$, the workload is balanced.

Note, that the configuration was made at compile-time to enable compiler optimizations. We use the following form to describe a concrete experiment setup: {count:COUNT, size:SIZE, host:HOST-TIME, rel.phi:PHI-RELATIVE-TIME, balance:BALANCE}. The HOST-TIME and PHI-RELATIVE-TIME were measured by evaluating the functor 10 times (in a single thread) and taking the average. The other values are directly specified in the experiment setup.

We have compared three approaches:

- `hybrid_for_each` from HyPHI with one or two coprocessors

variant	# Phi	time (s)	speedup	Phi util.
host-only parallel (TBB)	0	125.086	1	N/A
`hybrid_for_each`	1	9.332	13.4	93.88%
`hybrid_for_each`	2	5.455	22.9	82.28%
`pragma` offload	1	9.760	12.8	98.91%
`pragma` offload	2	5.901	21.2	83.70%

Table I
COMPARISON OF DIFFERENT FOR-EACH VARIANTS.
SETUP:{COUNT:10000, SIZE:4, HOST:0.279, REL.PHI:0.83, BALANCE:1}

- compiler's offload `pragmas` combined with TBB's `parallel_for_each` and static work distribution to run the work on one or two coprocessors (host is idle)
- host only execution of `parallel_for_each`

1) Total speedup: In this experiment, we compare the total execution time of the host-only TBB implementation to hand-written `pragma`-based versions and the HyPHI library. A synthetic benchmark code with the following configuration was used: {count:10000, size:4, host:0.279, rel.phi:0.83, balance:1}.

As shown in Table I, coprocessor-enabled versions can bring significant performance improvements for this benchmark configuration. This is due to the good suitability of the synthetic example configuration for the coprocessor (0.83 relative thread performance compared to host) and the high number of simultaneous threads (240) available on the coprocessors.

Even though this configuration favors the coprocessor, we observe additional performance gains over the `pragma` -based versions when the *hybrid_for_each* is used.

In addition, Table I shows coprocessor utilization (where applicable). This is reported as the percentage of time that the coprocessors executed the user's function without measuring idle time or other overheads such as data transfer. In this regard, we observe that even though the overheads with `hybrid_for_each` are slightly higher, the absolute performance improves due to utilization of the full system.

2) Processing time per item: The time necessary to process an item on the host is an important factor to be considered when offloading. If this time is very short, it may not be worth utilizing full hybrid execution. There is always some overhead associated with hybrid execution which needs to be amortized. In this experiment, we aim at finding the work granularity for which it pays off to use our library, by running the same experiment with different item processing times. The experiment setup was {count:10000, size:4, host:0.0015-1.3, rel.phi: 0.7, balance:1}.

As observable from Figure 3, the average time for processing one item does not have to be very high to beat `pragma` -based offloading. With our hybrid approach, the part of the implementation that does the host-side processing initializes very quickly and can start processing the data even before the coprocessor-side has been fully initialized.

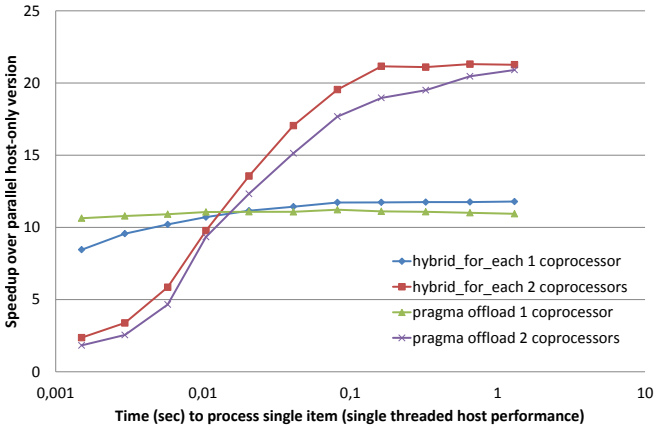

Figure 3. **Effect of the item processing time** – The hybrid approach can outperform `pragma`-based versions even for workloads that favor the coprocessor. Setup: {count:10000, size:4, host:0.0015-1.3, rel.phi: 0.7, balance:1}

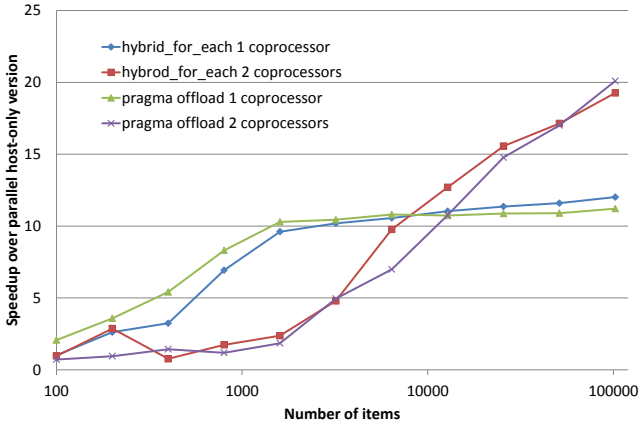

Figure 4. **Effect of the number of items** – for one coprocessor, 1000 items are enough to get significant speedup. To make the best use of two coprocessors, the sequence must be at least 20 thousand items long. Setup: {count:100-102400, size:4, host:0.013, rel.phi:0.74, balance:1}

3) Number of items: Offloading has some fixed overhead associated with the start-up and termination of the execution and some overhead proportionate to the number of items. For our library also the data serialization/deserialization of data might be an additional overhead.

In this experiment, we investigate the impact of the total number of items for performance. We want to derive insight about the total work amount required for using hybrid execution. The experiment configuration was {count:100-102400, size:4, host:0.013, rel.phi:0.74, balance:1}.

As can be seen in Figure 4, for most cases offload approaches (hybrid and pragma) work better when a higher number of items is computed. In this experiment, we have again chosen a good suitability for the coprocessor (rel.phi: 0.74). Even with this sub-optimal setup (for the host), our

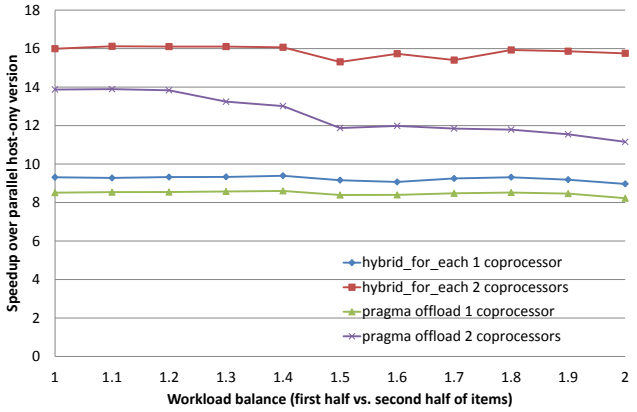

Figure 5. **Effect of processing time balance** – The hybrid approach can adapt to imbalances via dynamic work distribution. Setup: {count:10000, size:4, host:0.14, rel.phi:0.82, balance:1.0-2.0}

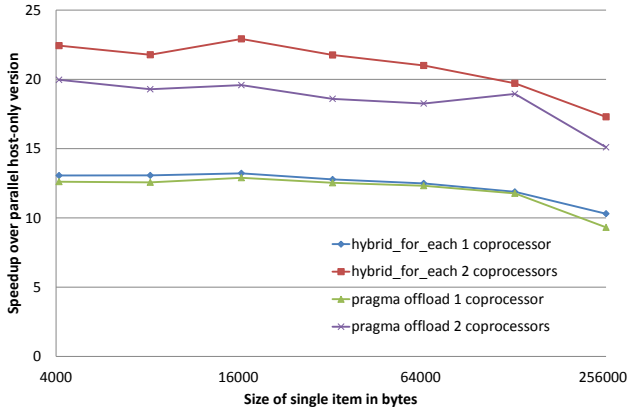

Figure 6. **Effect of the item size** – Setup: {count:10000, size:4k-256k, host:0.22, rel.phi:0.82, balance:1}

approach performs similar to the *pragma* version. In multiple cases, it even achieves better performance. This emphasizes the potential of hybrid execution.

4) Processing time balance: To process items on two coprocessors, we need to decide which items to process on each of the coprocessors. If `pragma` offloading is used, we do that by splitting the array into two equal parts. This works well, as long as the average processing time for both parts is about the same. However, this may not always be the case.

Hence, we have created an experiment where the first $n/2$ items take less time to evaluate than the second $n/2$ items. Within each half, the processing time is constant. Using `pragma` based offloading it is difficult to adapt to such situations at run-time.

Figure 5 shows experimental results for {count:10000, size:4, host:0.14, rel.phi:0.82, balance:1.0-2.0} setup. When 2 coprocessors are used with `pragma` offloading, we observe a performance decrease. This effect is less severe for our hybrid version featuring dynamic work distribution.

5) Size of item: The size of individual items may have effects on overall performance if the processing time remains constant. The `hybrid_for_each` data serialization, transfer and deserialization may introduce additional overheads depending on item size.

Therefore, we investigated different item sizes for the following setup: {count:10000, size:4k-256k, host:0.22, rel.phi:0.82, balance:1}. Figure 6 shows that despite the additional overhead, good performance can be achieved for sizes of up to 256k.

6) Relative PHI time: If the work vectorizes very well, it is possible to get much better performance per thread on the coprocessor than on the host. Considering the fact that the coprocessor can run a higher number number of threads, this performance advantage may become huge. In such situations there might be little room for improvement via hybrid execution.

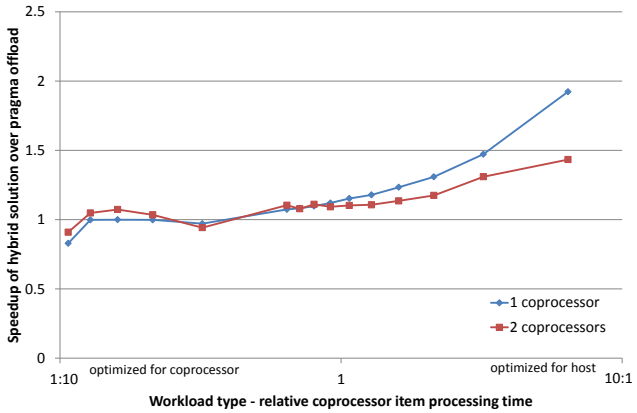

Figure 7. **Relative coprocessor processing time** – Hybrid execution outperforms `pragma` based version when the workload is favorable for the host. Setup: {count:10000, size:4, host:0.0155-1.55, rel.phi:6.4-0.64, balance:1}

On the other hand, there are situations where the host can process a single item much faster than the coprocessor. In such cases, hybrid or host-only solutions could significantly outperform pure offload solutions.

Therefore, we created an experiment with two extreme scenarios: code that can significantly benefit from vector units of the Xeon Phi coprocessor and code where the host has performance advantages. The setup was {count:10000, size:4, host:0.0155-1.55, rel.phi:6.4-0.64, balance:1}.

Figure 7 depicts the speedup of the hybrid approach compared to `pragma` based versions. We observe, that for this experiment performance is similar if the code is highly suitable for the coprocessor. *hybrid_for_each* provides better performance when the single thread processing time is lower on the host.

relative Phi time	relative time map : reduce	improvement of hybrid over `pragmas` 1 coprocessor	2 coprocessors
7.60	1 : 100	1.43	1.13
7.60	1 : 1	1.61	1.22
7.60	100 : 1	1.77	1.30
0.83	1 : 100	0.65	0.62
0.83	1 : 1	1.00	0.69
0.83	100 : 1	1.11	1.02

Table II
HYBRID MAP-REDUCE – FIRST TWO COLUMNS SHOW EXPERIMENT SETUP, LAST TWO COLUMNS SHOW IMPROVEMENT OF HYBRID IMPLEMENTATION OVER PRAGMA-BASED IMPLEMENTATION.

B. Hybrid map-reduce

The overall structure of the `hybrid_reduce` is similar to `hybrid_for_each`. Unlike `parallel_reduce` from TBB it uses iterators to specify input. This differs from the TBB range concept.

With the more complicated algorithm (two functors, the added complexity of reductions), there is a new important workload characteristic to consider – the comparison between the time it takes to perform single map operation (apply the first functor to a single input value) and the time it takes to perform one reduction (reduce two mapped values to one).

The experimental results for map-reduce are shown in Table II. The *relative reduce time* is the time it takes to perform a single reduction relative to the time it takes to perform single map operation. The last two columns show the speedup of the `hybrid_reduce` over `pragma`-based offload combined with TBB `parallel_reduce`.

C. Smoothed-particle hydrodynamics

Besides the synthetic benchmarks, we have also implemented a real-world physical simulation – the smoothed-particle hydrodynamics [2]. It is a simulation of a gas cloud in space (a nebula). For our purposes, the most important feature of the SPH is the fact, that the processing times are often greatly unbalanced because they depend on the density of the cloud – if there are many particles in some area, those particles take a lot of time to process.

Overall, the experiment could be approximately described as {count:1 000 000, size:112, host:0.005, rel.phi:8, balance:1-thousands}. Note that in this case, the imbalance is not between the first and second half of items (that would be 1 on average), but between the first and second half of items ordered by the processing time. The actual value depends on the concrete experimental setup and (more importantly) current state of the simulated universe. Also, the relative performance of Xeon Phi is an approximation, since the actual value largely depends on the concrete setup – the evaluation of a single item requires not only computation, but also significant (and variable) amount of memory access.

Performance results can be seen in Table III. Observed speedups are consistent with the approximated relative co-

configuration	time (seconds)	speedup
host only (TBB)	5185.37	1
1 coprocessor	2486.71	2.09
2 coprocessors	1533.14	3.38

Table III
SMOOTHED-PARTICLE HYDRODYNAMICS EXPERIMENT RESULTS, HYBRID VARIANTS ONLY

processor performance (rel.phi: 8). Hence, one could assume that the total performance of all of the host cores is slightly lower than the total performance of all of the coprocessor's cores. Ideally (without overhead and latencies), the speedup would be around 2.25 for 1 coprocessor and 3.5 for two coprocessors.

VI. RELATED WORK

The basic set of tools provided by Intel (the offloading compiler and SCI library) do not support hybrid execution directly. However, when combined with OpenMP or TBB to parallelize execution, they can provide good performance [3]–[5]. Hybrid execution needs to be implemented by the developer or a library. An example of such library is the StarPU [6] system. It provides task scheduling and execution mechanisms that can use the host, GPUs, and Xeon Phi. StarPU also takes care of data transfers. Compared to our library, it is a lower level solution, working with tasks rather than high-level parallel constructs.

There is another set of libraries available for Xeon Phi – high-level mathematical libraries like Magma [7] or MKL [8]. These libraries use the Xeon Phi internally to speed up the functions they provide to the user. These functions are often high-level, linear algebra (BLAS) operations and their implementation is highly optimized. On the other hand, the user only has a fixed set of functions that cannot easily be extended. In our approach, users provide their own code and data structures. As a downside, it can be much harder to achieve performance similar to specialized domain-specific libraries. Our approach is more suitable for a different set of use-cases where existing library functions are not sufficient but some common patterns (the algorithms) can be identified.

The C++ AMP [9] technology targets GPUs and, like our library, relies heavily on C++. However, there are major differences. The C++ AMP does not aim at full hybrid execution and it requires a more explicit accelerator programming. Still, it could probably be adapted for Xeon Phi and hybrid execution.

The Offload compiler developed by Codeplay supports offloading of parts of C++ applications to devices like the IBM Cell SPEs and GPUs, with the compiler taking care of the call graph duplication, functional duplication, and replication of global data [10]. Theses techniques are based on C++ language extensions. However, hybrid execution, as with our library, is not a goal.

Intel provides an MPI library for the Xeon Phi. It may be used to reach similar goals as our library. However, the architecture and coding techniques are completely different (message-passing C API rather than a C++ template library). With message-passing, the work distribution must be controlled by the user's code.

OpenACC [11] is a relatively novel standard for programming accelerated systems. It is based on source-code annotations similar to OpenMP directives. The approach enables the offloading of computational tasks from a host-CPU to an attached accelerator. It aims at providing portable accelerator programming. However, similarly to Intel's offload (Section III), automatic hybrid execution with multiple accelerators is not supported.

The HyPHI library is based on our previous work [12] but our old work was only partially based on tasks. Several management threads were used to support offloading. We identified this as an overhead. Apart from the serialization support, the new library does not share any sources with the old one. Besides the more task-centric approach, improvements have been made in multiple ways. For example, HyPHI features support for different hybrid strategies to enable performance tuning.

VII. Summary and Conclusion

The HyPHI library presented in this paper provides high-level support for parallel programming constructs such as *foreach* and *map-reduce* along the lines of TBB while being able to utilize all CPU cores and multiple Xeon Phi coprocessers within a hybrid parallel execution model. We implemented this library based on our experiences from previous work [12], using established C++ programming techniques as well as TBB task-parallelism whenever possible.

We demonstrated the ease of use of our library constructs by comparing it to existing programming approaches currently available for Intel Xeon Phi. Experimental results indicated the viability of our approach. We observed that hybrid execution can lead to significant performance improvements compared to parallel host-only and offload-`pragma` based codes. We evaluated a set of different performance-relevant parameters which help to identify cases when the HyPHI library should be used. In addition, we have shown that a computational physics SPH simulation application can benefit from our library and hybrid execution.

In future, we will further enhance the library with support for more hybrid programming constructs such as *hybrid_transform* or *hybrid_pipeline*. Moreover, we will extend the tunability of the library such that it can adapt to varying resource utilization and/or workload characteristics.

Acknowledgment

This work was partially supported by the European Commission's FP7, grant no. 288038, AutoTune. We thank Intel for providing software and hardware support.

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. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics - Theory and application to non-spherical stars," *Monthly Notices of the Royal Astronomical Society*, vol. 181, pp. 375–389, November 1977.

[3] L. Koesterke, J. Boisseau, J. Cazes, K. Milfeld, and D. Stanzione, "Early experiences with the Intel Many Integrated Cores accelerated computing technology," in *Proc. of the 2011 TeraGrid Conference*, ser. TG '11. New York, USA: ACM, 2011, pp. 21:1–21:8.

[4] A. Heinecke, M. Klemm, D. Pflger, A. Bode, and H.-J. Bungartz, "Extending a highly parallel data mining algorithm to the Intel Many Integrated Core architecture," in *Euro-Par 2011: Parallel Processing Workshops*, ser. LNCS. Springer, 2012, vol. 7156, pp. 375–384.

[5] A. Heinecke, M. Klemm, and H. Bungartz, "From GPGPU to many-core: Nvidia Fermi and Intel Many Integrated Core architecture," *Computing in Science Engineering*, vol. 14, no. 2, pp. 78–83, March-April 2012.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011.

[7] J. Dongarra, M. Gates, Y. Jia, K. Kabir, P. Luszczek, and S. Tomov, "MAGMA MIC," http://icl.cs.utk.edu/magma/software/index.html.

[8] Intel, "Using Intel Math Kernel Library on Intel Xeon Phi Coprocessors," http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_userguide_lnx/GUID-108D4F3E-D299-4D35-BAED-ECC5B48DA57E.htm.

[9] Microsoft, "C++ AMP: Language and programming model version 1.0," August 2012, http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf.

[10] A. F. Donaldson, U. Dolinsky, A. Richards, and G. Russell, "Automatic offloading of C++ for the Cell BE processor: A case study using offload," in *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*, ser. CISIS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 901–906. [Online]. Available: http://dx.doi.org/10.1109/CISIS.2010.147

[11] OpenACC-Standard.org, "The OpenACC Application Programming Interface - Version 1.0," http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, 2011.

[12] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, and B. Bachmayer, "Efficient hybrid execution of C++ applications using Intel Xeon Phi coprocessor," *CoRR*, 2012, http://arxiv.org/abs/1211.5530.