OCR-Vx – An Alternative Implementation of the Open Community Runtime

Jiri Dokulil, Martin Sandrieser, Siegfried Benkner
Research Group Scientific Computing
University of Vienna, Austria
(jiri.dokulil,martin.sandrieser,siegfried.benkner)@univie.ac.at

ABSTRACT

OCR-Vx is an implementation of the Open Community Runtime specification. It is being developed to provide an alternative to the implementation released within the X-Stack program. Two variants of OCR-Vx are available targeting distributed-memory and shared-memory systems including support for accelerators via OpenCL. Despite an early stage of development, our first experiments show that a properly designed application can achieve very good performance on top of OCR-Vx for small-scale systems. We have also designed several extensions of the OCR API, to better support heterogeneous systems, file IO, and improve performance of some OCR operations in a distributed environment.

1. INTRODUCTION

The extreme scale and increased performance variability of future high performance computing systems pose many new challenges to parallel programming models and runtime systems. The Open Community Runtime (OCR, [15]) is a recent effort for a runtime system for extreme scale parallel systems. Key features of the OCR are the event-driven, asynchronous task-based approach for expressing parallelism and its support for fault-tolerance. Similar to other task-based systems, the OCR does not give the developer direct control of what is being executed by each thread or process at any time, but rather lets the developer express the parallelism available in an application by creating a large number of tasks, which perform parts of the computation. Synchronization is expressed by specifying dependencies between tasks.

We have implemented the OCR standard in a shared memory environment [8], providing an alternative to the implementation crated by the consortium responsible for the OCR specification. Based on our shared memory implementation we are currently working on a version that supports distributed memory systems including parallel accelerators. Furthermore, we have designed a platform description language in order to provide an explicit representation of the target platform that can be utilized by the runtime and by applications.

The rest of the paper is organized as follows. First, we briefly introduce the OCR in Section 2, provide an overview of the main issues faced by OCR implementers, and finally describe how these issues are solved in both of our implementations (shared-memory and distributed memory), including the extensions which allow the OCR to target heterogeneous architectures. Section 3 provides a brief overview of related work. Section 4 describes the applications which were used to test our implementations. The experimental results are shown in Section 5. Section 6 briefly introduces several OCR extensions which we are currently working on, but which have not yet been implemented in a way that would allow experiments to be carried out. Section 7 concludes the paper.

2. OPEN COMMUNITY RUNTIME

The Open Community Runtime is a work-in-progress effort to create a low-level, task-based runtime for extreme-scale parallel systems, with fault-tolerance support. The basic idea of the OCR is to express the computation using tasks referred to as *event driven tasks* (EDTs) and task dependencies, while organizing all data in *data blocks* (DBs) which are managed by the OCR runtime. The task approach should help the computation dynamically adapt to large scale systems with high variability of performance. With the data blocks and their explicit linking to tasks through dependencies, the runtime is able to move a task to a different node or restart the task on a different node (with the original data) should a node fail.

2.1 Overview of the OCR concepts

The execution of tasks is controlled by dependencies with the help of *events*. A dependence connects a *post-slot* of an event to a *pre-slot* of a task or another event. This defines a control dependence. When an event is triggered, it notifies all objects connected to its post-slots. A task may start once all its pre-slots have been notified (*satisfied* in OCR terminology). There are several types of events in the OCR and the exact rules defining when the events trigger depend on the event type. There is one event associated with each task. This event is triggered when the task finishes. It is possible to mark a task as a *finish* task, in which case the associated event is triggered once the task and all its children (child tasks) created directly or indirectly by that task have finished.

The same dependence mechanism is also used to define data dependencies. There are two options. First, it is possible to use a data block as an origin of a dependence, which directly satisfies the pre-slot. Second, a data block may be passed along a control dependence. A task may return a data block, which is then passed to the event associated with the task. The event then sends the data block to all tasks and events bound to its post-slots when it notifies (satisfies) their pre-slots.

A task can only access data in a data block if it created the data block or the data block was assigned to one of its

pre-slots. As a result, the runtime is aware of all the data objects the task can possibly access. There are four different access modes, which control the way data blocks can be concurrently accessed by multiple tasks. An access mode is specified when a dependence is set up and it is valid for the data block which is passed along that dependence. The access modes are: constant (CONST; read-only coherent), read-only (RO; data may be concurrently modified by other tasks), read-write (RW; non-coherent access, possibly writing to the data), and exclusive write (EW; coherent, needs to be the only writing task). The OCR runtime may create multiple copies of a data block to satisfy the access modes, for example to allow one task EW access to the original, while multiple tasks have CONST and RO access to the copy. The non-coherent modes provide some basic guarantees regarding reads and writes, but it is mainly the programmer's responsibility to maintain consistency of the data in case of concurrent access.

2.2 Major Issues in Implementing OCR

At the moment, we maintain two implementations of the OCR. OCR-Vsm is a shared-memory implementation, OCR-Vdm supports distributed-memory systems. Together, they are referred to as OCR-Vx. Both variants use the Intel Threading Building Blocks (TBB) library to maintain a pool of threads and map local tasks to threads. Since TBB only supports shared-memory architectures and very simple task dependence mechanisms, a significant layer of library code has to be provided between the OCR API and the TBB. Table 1 provides an overview of the key responsibilities of this layer. All of this functionality is provided directly by the OCR-Vx implementations, except task execution support, which is provided by the TBB library.

The TBB library was selected because it provides very lightweight and efficient tasks. We sometimes use non-TBB threads to do certain types of work, which is a scenario that the TBB was specifically designed to support. Furthermore, the tasks in the TBB are easier to control and more flexible compared to OpenMP tasks.

shared-memory OCR-Vsm	distributed-memory OCR-Vdm
management of OCR objects	
management of objects references (GUIDs)	
	remote object creation
management of dependence graph	
local scheduling of tasks (using TBB)	
acquisition of DB locks by EDTs (local locks)	acquisition of DB locks by EDTs (local and remote)
	data coherency across nodes
	handling of messages

Table 1: Key responsibilities of OCR-Vx variants.

2.3 Shared-Memory Implementation

In September 2014, the OCR working group released version 0.9 of the interface specification. We have created an OCR implementation which complies to the 0.9 specification, with the exception of some of the experimental and extension APIs. After the 1.0 specification was published in June 2015, the implementation was updated to match the new version, renamed from OCR-TBB to OCR-Vsm and released as open source. The implementation supports only shared memory multi-/many-core systems (conventional multicore CPUs and the Xeon Phi coprocessor) and is built on top of the Intel Threading Building Blocks (TBB, [14]) library, most notably the tasks scheduler provided as part of TBB. The following paragraphs deal with the different aspects of the OCR-Vsm implementation, as categorized in Table 1.

Management of object identifiers.

Each OCR object is represented by a globally unique identifier (GUID). We use the address of the corresponding object in memory (the pointer) as the GUID. The GUIDs are in effect assigned by the dynamic memory management when the object is allocated and freeing the object also frees the GUID for future use. With this approach, mapping from GUIDs to objects is very fast. It does not allow the OCR objects to be enumerated, like they could be if the GUID was for example a unique sequential number. The runtime does not need such enumeration (see the next paragraph for details), but it could be useful for debugging.

Management of OCR objects.

The way the OCR API is constructed, it is not necessary to maintain a repository of OCR objects. The runtime gets all the necessary information from the user's call to the API and from local data of the OCR objects that are being processed. Therefore, it is not a problem that the pointer-based GUIDs don't support any way of accessing the OCR objects without their GUIDs.

To give a more concrete example, when an event is created, the user receives its GUID (pointer). To be of any use, the event has to be used as a destination of a dependence or satisfied by an explicit call to the OCR API. In the first case, the source of the dependence will store the GUID (the pointer) to the event, so that it can trigger the event when needed. In the second case, the GUID is provided by the user in the API call. The runtime never needs to search for the event without being provided the GUID.

Management of dependence graph.

The dependence graph is conceptually represented using the list-of-neighbors approach. Each node (event or task) has a list of post-slots. When a tasks finishes or an event is triggered, it goes through the list of its post-slots and satisfies the corresponding pre-slots of the tasks or events that depend on it. All of this is performed by directly calling methods of the objects that represent the tasks and events in the runtime. If as a result of such call a task becomes ready to run (*runnable* in OCR terminology), it tries to acquire (lock) all data blocks that have been assigned to its pre-slots. Once all data blocks have been acquired, the task becomes *ready* and is submitted to the TBB scheduler.

Note that a significant amount of time may pass between the moment a task becomes runnable and the moment it is ready for execution. The process of data block acquisition is detailed in one of the following paragraphs.

Local scheduling of tasks.

Ready tasks are submitted to the TBB scheduler (*spawned* in TBB terminology). The scheduler uses a fixed thread pool and task stealing to run the tasks. By default, the TBB

thread pool contains as many threads as there are CPU cores available. Even though TBB also provides means to define dependences between tasks, we only submit the tasks that can be immediately executed. The TBB task dependence mechanism is not used, since it is too simple to support the OCR dependence graph and data block acquisition.

Once a task finishes, the event associated with the finalization of the task is satisfied. The satisfaction signal may further propagate through the dependence graph, based on the OCR rules. All data objects held by the finishing task are released, which may cause other tasks to become ready (see the next paragraph for details). All of this is handled in the task epilogue, forming an event-driven dependence tracking system, where no threads are being suspended while waiting for data to become available. The threads only hold fine grained locks which are used to control access to the structures maintained by the runtime and are only held for a brief period, when the structure is being updated.

Acquisition of DB locks by EDTs.

As discussed above, an EDT becomes runnable once all its pre-slots have been satisfied. However, some of the pre-slots may have a data block associated with them and that data block may still be used by other tasks. Each pre-slot also specifies an access mode, which has to be combined with the access modes used by all other tasks that use the data block to determine whether the data block can be acquired by the ready task or whether the task needs to wait. In our implementation, each data block tracks the number of tasks that have acquired the block using a certain access mode. With four modes, we have four such numbers per data block to determine whether the data block can be acquired using a certain access mode. For example, if the data block has been acquired by two tasks in constant mode, it cannot be acquired using exclusive write or read-write modes.

If all data blocks needed by a task can be acquired, the task is ready and is submitted to the TBB scheduler for execution. If a data block cannot be acquired, the task is added to the block's list of waiting tasks. This situation can only be encountered if the data block is currently being used by at least one other task. When a task finishes, it releases all acquired data blocks. It also checks the waitlists for tasks that could acquire any data block that has just been released by the task. This can result in one (or more) task becoming ready and being submitted to the TBB scheduler. Access to the objects that form the dependence graph is protected by fine-grained locks. These locks, but also the data blocks acquired by a task, are always acquired according to a global ordering of the locks to avoid deadlocks.

Intel Xeon Phi.

An important aspect of our design is the fact that there is no central coordinator or other entity responsible for data block acquisition and dependence graph management that would imply the use of a highly congested, global lock. This is important for manycore systems, like the Xeon Phi, where such a lock would become a serious performance bottleneck, not only due to serialization of execution but also due to high cost of maintaining such lock through cache coherency mechanisms. In our experiments, we were able to achieve similar performance as OpenMP, which is the technology recommended (and significantly tuned) by Intel for the Phi.

2.4 Distributed-Memory Implementation

Based on our shared-memory version, we are implementing OCR for distributed-memory systems, possibly equipped with accelerators. While the basic ideas are the same, the distributed OCR-Vx (OCR-Vdm) does not use the shared-memory implementation locally within the node. The reason for this is that the OCR-Vsm uses several optimizations that are not possible in a distributed-memory setting. The key differences are discussed in the following paragraphs, again following the structure shown in Table 1.

Management of object identifiers.

In the distributed-memory environment, the pointers are no longer unique. Different objects could be allocated at the same address on different nodes. Therefore, we include a node identifier in the GUIDs. Since we wanted to keep the GUIDs as 64bit numbers, it is not possible to also store the pointer to the object in the GUIDs, since the pointers (on the supported architectures) are also 64bit long. Therefore, all objects on a node are given unique (sequential) numbers, which are combined with the node identifier to form GUIDs.

However, the intention of the OCR consortium is to also support larger GUIDs, which would allow us to use the combination of node identifier and pointer to identify the objects.

Management of OCR objects.

Each OCR object is stored on one node of the distributed system. This node is determined when the object is created and the identifier of the node is encoded in the object's GUID. The second part of the GUID is a local identifier (from a numerical sequence maintained locally on each node) of the object within the node. As a result a local object directory is needed on each node to translate the sequential number to the actual address of the object.

Note that while each object is stored on one node, multiple nodes can hold a copy of a data block's data. The management of these copies is explained in a separate paragraph.

Remote object creation.

Because a task is executed where its corresponding OCR object is stored, it is necessary to allow remote creation of tasks. Otherwise, all tasks would be created and executed on the node that starts the first task (called *main EDT*). Using the experimental affinity API of the OCR, the user may specify the node where the task should be created in the ocrEdtCreate call. This call returns the GUID of the task, which consists of the target node identifier and the current object sequence number from the target node. As a result, the call entails communication with the target node, blocking the task. To minimize these delays, we do not obtain just one sequence number, but a whole block of such numbers and cache them for future use.

Management of dependence graph.

The main difference from the shared memory version is the fact, that the dependence graph is also distributed. The events and tasks that form the graph are distributed among the nodes. Therefore, it is no longer possible to propagate changes in the graph by calling methods of the objects. Instead, the updates are realized through messages, which can

be processed locally, if the affected object is on the same node, or sent over the network otherwise.

Local scheduling of tasks.

Since the task cannot be migrated after creation, the local task scheduling works the same as in the shared memory version. The important difference is in the way data blocks are acquired, since they are stored in a distributed memory, so a node may not always have the data or the data may not be up-to-date.

Acquisition of DB locks by EDTs.

Locally, the data blocks are acquired using the same mechanism as in the shared memory version, using acquisition counts per mode and waitlists. The important difference is the need to provide access to remote data blocks and maintain data coherency across nodes. These mechanisms are mostly separate from data block acquisition. If a data block needs to be acquired, but the node does not have a copy or a writeable copy, it uses the data coherency interface to request the copy. Until the copy is available, the data block cannot be acquired. Furthermore, the coherency algorithm may request that tasks should no longer be allowed to acquire the data block, if the local copy needs to be invalidated.

Data coherency across nodes.

Coherency of a data block is maintained by its owner – the node where the DB was created and which is encoded in the data block's GUID. This nodes maintains a list of valid copies, allows new nodes to acquire a (possibly writeable) copy, invalidates copies, and arbitrates between concurrent request for access to the data block from multiple nodes. At the moment, we use a simple coherency protocol, which guarantees the release consistency required by the OCR specification, but only uses a simple first-come-first-serve protocol for arbitration among nodes. As a result, if two tasks on node A and one task on node B need EW (exclusive write) access to a data block, it is possible that one task from A will be served, but before the second task can run, the updated copy is sent to node B, where the next task gets the data block. Finally, the data is copied back to A, allowing the second task to run. This could be improved, as the owning node does have all the information at some point in time. However, it is not as simple. The tasks may have other dependences than just the data block from the example.

Handling of messages.

Most OCR actions are handled using messages. For example, when an EDT is created, a new message is created, containing the parameters of the creation call. When an event is triggered, a message is created for each post-slot of the event, to satisfy pre-slots of the corresponding objects. The GUID of the affected object is used to determine the destination of the message, because the GUID encodes the identifier of the node where the object is located. The message is then sent to a queue on the receiving node. The messages in the queue are processed in the order in which they arrive. However, this is not sufficient to ensure the consistency model prescribed by the OCR specification. If two m_1 and m_2 messages travel to two different nodes N_1 and N_2, they may arrive in different order. The actions triggered by these messages may result in sending of further messages m_1' (based on m_1) and m_2' (based on m_2). If both m_1' and m_2' are sent to the same node N_3, they may arrive and be processed in any order, irrespective of the initial ordering of m_1 and m_2, since the ordering is only preserved among messages received from the same node. As a result, m_2' may be processed before m_1'. In certain situations, this may constitute a race condition, leading to incorrect results. We have several mechanisms in place to prevent this situation. However, under some rare circumstances (none of which are encountered by our test applications), these may not be enough. We already have the design for an alternative message handling system, which will always provide correct results. This will be available in the next version of OCR-Vdm.

Intel Xeon Phi.

As mentioned, the shared memory OCR-Vx supports both traditional multicore CPUs and Xeon Phi coprocessors. However, it is not possible to run the distributed implementation on the host CPUs and the coprocessors at the same time. The reason for that is, that the work performed by a task in the OCR is specified by a pointer to a function. This pointer must be meaningful across all nodes. As the Xeon Phi requires a different binary, the address would likely not be the same. In the offload mode, the Intel compiler manages the translation automatically, so it could be used to run on the host and the Xeon Phi at the same time, but our current implementation does not support the offload mode. It may be included in the future, but the introduction of the bootable Xeon Phi processors (which are deployed the same way as normal CPUs and no longer use the work offloading model) makes the support for offloading to the coprocessor cards less important.

2.5 Support for OpenCL

The OCR's architecture of tasks, dependences, and explicit representation of data in data blocks shares some similarities with the concept of kernels in OpenCL. The invocation of a kernel is akin to a task invocation in OCR, the buffers correspond well to data blocks, and the fact that a kernel's execution should be independent of other concurrently executing kernels is similar to the requirement for non-blocking EDTs in the OCR. Also, OpenCL events are similar to the OCR events and it is possible to specify a list of events (a waitlist) that have to finish before the kernel starts, which resembles the pre-slots of an EDT. Therefore, we decided to try combining OCR and OpenCL. Since OpenCL kernels have no or limited ability to control the overall execution, OCR is the clear candidate for the primary programming model. Therefore, we have extended the OCR API to allow OpenCL kernels to be used as a special kind of OCR tasks. Special calls, which we have added to the API for this purpose, are used to create a task template (compile the kernel source) and create the tasks (which includes the ND-ranges that define the global and local work sizes). Once an OpenCL task has been created, in can be used as any other task – it can be used as both source and destination in dependences. The data blocks assigned to the tasks pre-slots are transformed into OpenCL buffers and then passed to the kernel. This means that the kernel's arguments have to match the pre-slots of the task.

Internally, the OpenCL sub-system is managed by a sin-

gle, dedicated thread. It would be possible to implement an OpenCL task by a normal OCR task that would call the kernel inside its code, but that would block the TBB thread that was used to run the task. Such blocking is undesirable in TBB, since the blocked thread cannot be replaced by a new worker thread and as a result, there would be less threads than CPU cores, resulting in core underutilization. In our solution, the extra thread creates oversubscription, but since it spends most of the time blocked inside an OpenCL call or while waiting for OpenCL tasks to execute (this is also done using a blocking call, rather than active waiting), it is not a problem.

At the moment, the OpenCL extension is only available in the OCR-Vsm. Furthermore, the implementation does not reuse OpenCL buffers, so if the same data block is used by two consecutive kernel executions, it is not preserved on the device, resulting in unnecessary data movement. This can be fixed using the same mechanism that the OCR-Vdm uses to maintain data block copies. Similarly, current design can also be used to provide OpenCL in the OCR-Vdm, we just didn't implement it yet.

3. RELATED WORK

There are several run-time systems based on dynamic task management, including StarPU [1, 10], the Nanos++ runtime used in OmpSs [4], the Charm++ system [12], and HPX [11]. Despite being based on the same idea, they differ from the OCR in many design choices and in the features they provide. This makes direct comparison hard or impossible.

Since the current OCR reference implementation[1] focuses on correctness, rather than performance, we have not used it for performance comparison. Instead, we compare the performance of our test applications to non-OCR implementations of the applications, usually OpenMP. Our implementation of OCR shares no source codes with the official reference implementation of OCR except for the OCR headers that define the API. We have extended the headers to also support OpenCL tasks. These extensions are not supported by the official implementation. Except these extensions, the two implementations are compatible, and it is possible to compile an application and link it to either implementation.

4. APPLICATIONS

For the experimental evaluation, we have selected two relatively simple simulation applications. The Seismic code is used to evaluate the distributed implementation, providing a comparison with parallel OpenMP code. The Levenshtein application demonstrates the OpenCL extensions. All codes (including the OpenMP-based parallel Seismic) were hand-tuned for the experimental machine to give a fair comparison.

4.1 Seismic

The seismic application was inspired by the seismic example distributed as part of the TBB library. It simulates propagation of seismic waves through 2D terrain. There are several properties associated to each grid point (stress, velocity, dampening, ...). Seismic runs in several iterations and each iteration comprises three phases. First, the initial seismic pulse is updated, which involves updating velocity of a single point. Second, horizontal and vertical stress is updated for each grid point based on values of properties (other than the stresses) of the point and its neighbors to the right and below. Finally, the seismic wave velocity is updated for each point based on properties (not including the velocity) of the point and its neighbors to the left and above. There are no data dependences within a phase, just between phases and only among neighboring grid points.

In both variants (OCR and OpenMP), the parallelization is performed by processing rows in each phase independently. This entails two parallel `for` loops in the OpenMP variant. For the OCR variant, we have used a much more complex (and potentially more efficient) strategy. The data is split into several blocks, each containing the same number of rows (except, possibly, for the last one). These blocks are distributed to different compute nodes that do not have shared memory, requiring much less data movement, as only the bordering blocks would have to be synchronized. To provide a reasonable trade-off between the number of data blocks and the available parallelism, each block is processed in parallel by multiple tasks. The dependences between phases are done in a fine-grained fashion to allow parallelism not just within a phase, but also between phases. Details are given in [8].

4.2 Levenshtein

The Levenshtein application computes the Levenshtein edit distance of two strings. We use the basic version, which evaluates the whole matrix of all partial edit distances for initial runs of both strings. The algorithm has significant data dependences. To compute a value at any position in the matrix, we need to know the values of neighbors to the left, upper-left, and up. This makes parallelization of the code much more difficult than in the case of Seismic. We exploit the fact that values on a diagonal can be computed independently of each other. This allows the algorithm to be parallelized, but makes it difficult to efficiently create the necessary OCR tasks, set up their dependences, and provide them with the necessary data. We only store the values of the matrix as long as they are needed for further computation, which further complicates the data management.

The Levenshtein algorithm has an important property: If the whole matrix is divided into equally-sized, rectangular blocks, the dependences between the blocks are exactly the same as they are for individual values – a block depends on the blocks to the left, upper-left, and up. We use a two-level approach in the Levenshtein application. On the first level, larger blocks are computed in parallel. To compute each block there are two implementation variants. First, we use the same parallel approach as on the first level, this time on much smaller blocks. Each (small) block is then processed by serial code. Second, an OpenCL kernel computes the whole block. These implementation variants are selected statically. All blocks may be evaluated on the CPU, on the GPU, or a statically determined combination of the both (e.g., 1 block on the CPU, 2 blocks on the GPU).

5. EXPERIMENTS

We have performed several experiments, using both the shared memory and distributed versions of the OCR-Vx. Some experiments with OCR-Vsm have been published in our earlier work [8] focusing on the Xeon Phi. We present here updated results (using the latest version of OCR-Vsm)

[1] `https://github.com/01org/ocr`

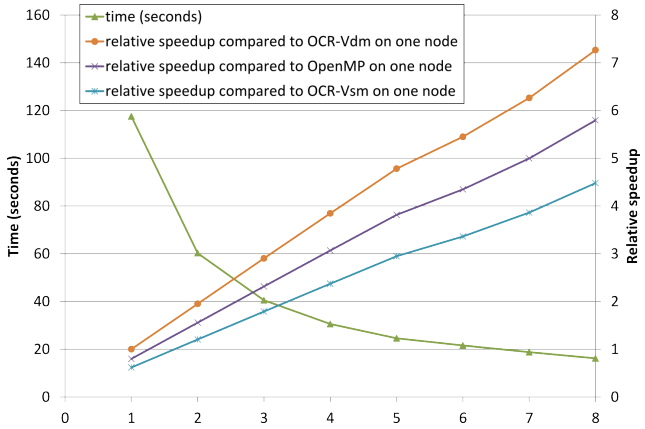

Figure 1: Performance and scaling of the OCR-Vdm when running the seismic simulation on up to eight nodes of a cluster. OCR-Vsm runs exactly the same code as OCR-Vdm, but the OpenMP variant is built from a separate (but equivalent) source code.

	OpenMP	OCR-Vsm	OCR-Vdm	
			1 node	8 nodes
host	93.75	72.50	117.57	16.18
Xeon Phi	35.34	35.53	N/A	N/A

Table 2: Wall-clock execution times (in seconds) of different variants of the Seismic application.

where they may provide additional insights.

The first experiment aims at exploring performance and scaling of the distributed version using the Seismic application and a cluster with 8 nodes (each with two 8-core Xeon E5-2650 CPUs), connected vie Ethernet. The results of the experiments are shown in Figure 1. The speedup achieved on 8 nodes is 7.27 when compared to the same code restricted to just one node and 5.80 when compared to OpenMP. Performance of Seismic using OCR-Vsm (using the same application code as OCR-Vdm) is even better than the OpenMP variant, although on the Xeon Phi they perform about the same. We have not identified the reason for the better performance of the OCR-Vsm variant, but we suspect the reasons are better utilization of cache locality and elimination of barriers between phases. The actual execution times are shown in Table 2. In all configuration (OCR-Vdm, OCR-Vsm, and OpenMP), all cores of each involved node are used to run the computation. Note that already with two nodes, the distributed OCR version is faster than the OpenMP version on one node (the speedup is 1.57) as well as OCR-Vsm (speedup 1.20). An execution trace of the program on 8 nodes by OCR-Vdm is shown in Figure 2.

The second experimental application is Levenshtein. This time, OCR-Vsm with OpenCL support was used, running the tests on the same machine, but also using the NVIDIA K20m GPU. This version is an early stage of development, but it already demonstrates the feasibility of combining OCR tasks with OpenCL kernels. An execution trace is shown in Figure 3. With CPU tasks only, the execution takes 70.51 seconds, while pure GPU execution needs 22.68 seconds to finish. Note that the GPU version is still done using the

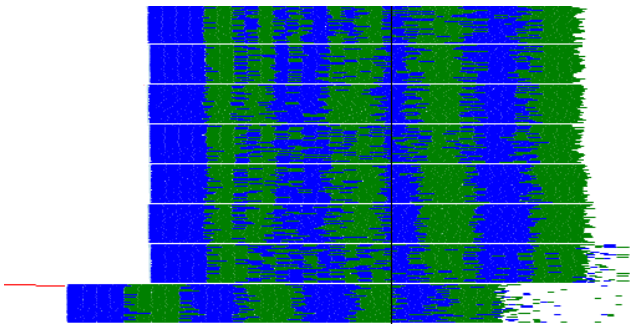

Figure 2: An interval of the execution trace of Seismic code (10 iterations), using OCR-Vdm on 8 nodes. The bars represent tasks and each row represents one thread (there are 32 threads per node). The different colors represent the different phases of the seismic algorithm. The 8 large blocks of rows correspond to the 8 nodes. The last node is the master (the initial task runs there), which has the original copy of the data and can therefore start earlier. The black line shows where the middle of the image was cut to conserve space, showing only the data for 4 of the 10 iterations. The 6 iterations that were cut just repeat the pattern. As you can see, the overall utilization of the cluster is very good (there are few white spaces in the trace), except for the slow startup of non-master nodes, caused by relatively low performance of the Ethernet interconnect, and the low utilization of the master near the end of the computation, caused by the early start of work on the node. With larger number of iterations, the effect on the overall performance would decrease.

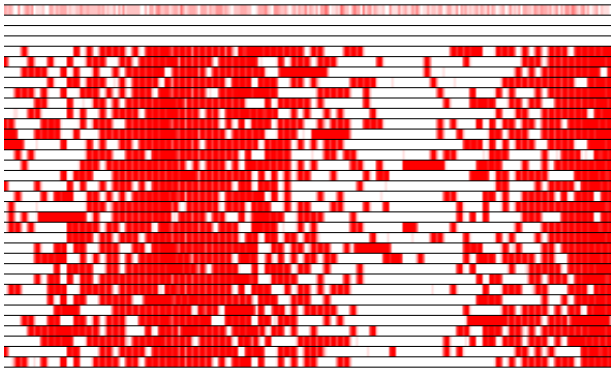

Figure 3: An interval of the execution trace of Levenshtein edit distance executed using OCR-Vsm on a GPU (first row) and 32 worker threads (rows after the gap). Each red bar represents execution of a task or OpenCL kernel. The white spaces represent the time the HW (GPU or a core) was idle. The overall utilization of the system is reasonable, but it is obvious that the CPU and GPU executions are far from being perfectly balanced.

OCR, but the tasks that perform the actual computation of the edit distance are executed on the GPU using the OpenCL extension.

Utilizing both the CPU and GPU results in an execution time of 28.65 seconds, demonstrating that it is not easy to balance execution on both at the same time, especially since we are using a static work allocation. It should be possible to improve, but the application would need a significant redesign to better distribute the work between the two. Since OpenCL tasks are separated from normal OCR tasks, the runtime cannot redistribute them automatically, without some kind of task variant support. Whether a task executes on the CPU or using OpenCL is determined by which function was used to create the task. It cannot be changed dynamically, for example in response to current resource utilization. If the runtime had the information that work of the OpenCL task could also be performed by a CPU task and vice versa, it could switch to the other variant and achieve better hardware utilization.

In its current form, the code is not suitable for the Xeon Phi. Due to limited parallelism, it takes 114.67 seconds.

6. OCR EXTENSIONS

Besides the OpenCL support, we are working on several other possible extensions for the OCR specification. Some of these extensions aim at enabling the exchange of performance-relevant information between an OCR runtime and an application, possibly improving overall performance. Other extensions provide new functionality that is not available in the current API. The proposed extensions (except for platform descriptions) are detailed in [7].

6.1 Platform Descriptions

The OCR aims at providing a unified runtime-system specification applicable for a wide range of computing systems and application scenarios. It does so by decoupling program representation (directed acyclic task-graph) from concrete hardware requirements wherever possible. Therefore the execution model employs a highly generic view of computational and storage resources which is built on *network connected nodes* that consist of *processing elements* with private *memories* and a globally accessible *shared name space* of OCR objects. [15, p. 11]

How the abstract entities such as *node*, *processing element* and *memory* are mapped to concrete hardware of the execution environment seems to be entirely up to an OCR implementation. While this ensures portability for OCR-conform applications, OCR implementations need to incorporate target-system specific optimizations to provide high performance. For applications that can benefit from target-specific information (e.g., hardware- locality) it is currently unclear how such knowledge can be efficiently exchanged between an OCR implementation and the application programmer. For example, information about extent and mapping of *affinities* (e.g., see [15, p. 35]) can not yet be easily queried. We belief that for many applications (i.e., featuring irregular computational patterns) this knowledge might be required either by the application programmer or an intelligent scheduling system. However, providing lower-level target specific information while still maintaining OCR's desired generality is challenging.

We have previously developed a platform description language (PDL) [17, 18] that aims at bridging the gap between

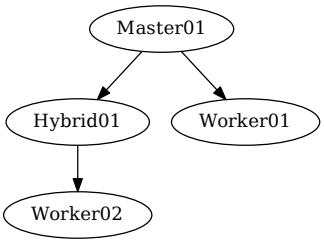

Figure 4: The OCR execution model described with the XML-based PDL.

generality and providing low-level performance relevant information by capturing both: (I) high-level platform usage patterns imposed by a programming system's execution model and (II) lower-level properties of concrete hardware and software resources. This was used to store and exchange information about hardware and software properties in heterogeneous computing environments to support code-generation for such systems. The PDL is based on a platform model that employs control-relationships between *processing units (PU)* as its main layer of abstraction. In addition, data-transfer and communication abilities are represented by *Interconnect* entities. *Memory regions* denote storage facilities that can differ in type and performance. Each of those entities can further be described with key-value pairs for properties which feature fixed keys for different types of entity incarnations.

In what follows we give an example on how to represent the OCR platform and execution model described in [15] with the PDL. Figure 4 depicts the abstract OCR execution model as described in [15] in generic coarse-grained PDL. The XML-based PDL descriptor only comprises the required control-relationships (arcs) between processing-units (nodes) and communication facilities (not depicted). By means of the PDL, an OCR computation (*mainEDT*) starts at the *Master* processing-unit. Additional computations can be executed on processing-units that act as *Hybrid* entities (i.e., can spawn additional tasks) or *Worker* entities (i.e., do not spawn additional tasks). This high-level descriptor aims a capturing the OCR execution model and does not (yet) comprise any mappings of the PDL entities to concrete hardware resources. Such mappings depend on a concrete runtime-system implementation. We see the PDL as a way for making mappings explicit to tools and users. This aims at answering, in a machine-processable way, the question of how system resources are employed by a concrete runtime implementation.

We also envision the use of PDL descriptors for making the runtime-system itself more portable. In the future, we will investigate a runtime implementation that can be parametrized by a system and/or application specific PDL descriptor. This might allow incorporating expert knowledge about an underlying architecture without restricting generality and portability.

6.2 Local identifiers

Ideally, it should be possible to process all OCR API calls (made by tasks) locally, without the need to send a message to a remote node and wait for the reply. With relaxed requirements for error detection, this is possible for most API calls. For example, the `ocrEventSatisfy` call can be

handled by a message that would inform the event that its pre-slot has been satisfied. The call only affects the satisfied event, not the calling task. Therefore, the function can return immediately, without waiting for the reply.

Functions which create objects are an exception. They have to return the GUID of the newly created object. Some implementations may need to contact another node, for example to get the current value of an ID sequence of the node where the object will actually be created. Once the GUID is determined and returned to the calling task, it may for example store the GUID in a data block, so that other tasks can read it and use it in further OCR API calls. However, it is also common that the GUID is only used to make several calls to the OCR API by the task that created the object. The GUID is not stored in any data block. A typical example would be when a task is created and all of its dependences are immediately defined by the creating task.

If the GUID is not saved, it is not necessary to have a true GUID – a *globally* unique identifier. Therefore, an identifier with a local validity may be sufficient. Since the OCR abstracts nodes, the only applicable scope besides the global scope is a task. Therefore, the OCR call which creates an object may return a *local identifier* (LID), which may only be used to make OCR API calls by the task that created the object. Such a LID cannot be saved to a data block and used by another task.

A different view of the LIDs is to view them as futures. Instead of returning the new object's GUID, a future is returned. The future may be used to make further OCR API calls, but the runtime does not yet send out messages that implement these calls. At some point in time (possibly after the task has finished) the future will be resolved, allowing the runtime to update the messages, providing the actual GUID instead of the future.

6.3 Labelled GUIDs

The OCR specification draft 1.0.1 contains a proposal for labeled GUIDs. The idea is to provide a convenient way of managing a large number of GUIDs without the need to pass all such GUIDs from task to task. Instead, a map is created. The user can then obtain a GUID from the map just by providing a GUID of the map and coordinates of the object within the map. There are two issues connected to the creation of objects in such maps. First, it may not be possible to create a GUID before the parameters of the creation call are known. Second, the map should be able to deal with concurrent requests to acquire the same object from the map. Since it should be possible to create and destroy objects in the map as needed, not just when the map itself is created and destroyed, the map should take care of all necessary synchronization and make sure that objects are created only once.

When a map is created, the user specifies a *creator* function, which is responsible for creation of objects in the map. Initially, the map is empty and no object is created. A task may then request an object at a specified index from the map. If the object does not exist, the map calls the creator function and the object is created. However, this creation has to be properly synchronized by the runtime, to make sure that concurrent calls with the same index return the same object. However, we would like all OCR API calls to return immediately, without any communication. Since synchronization without communication is not possible, we use LIDs to solve the problem. When an object is fetched from the map, the call returns a LID of the object. This can be done locally. If two tasks get the same object, they get different LIDs, but once resolved, the LIDs will point to the same object (same GUID).

In this case, providing LIDs instead of GUIDs is not a big limitation, since it's not necessary to send reference to such objects using GUIDs. Any task can fetch the identifier from the map and work with the LID.

6.4 File IO

At the moment, the OCR API does not deal with file IO. In this context, an important issue that has to be solved is resilience. The OCR is supposed to recover in the case of component failure. To do this, information can only be preserved outside of a task in a data block, which is under full control of the OCR runtime. If normal IO functions were used, the tasks would have side effects, which could lead to incorrect results if the tasks are restarted by the runtime.

An alternative, which we believe would provide a better match for the resilience requirements, is to create an alternative of memory mapped files – data block mapped files. With such extension, it would be possible to map a data block to a section of a file. When passed to a task, the data block would contain data from the file. If the data block is changed by a task, the changed data is eventually (it may have to be delayed to facilitate the required resilience) written back to the file.

7. CONCLUSION

With a properly designed application on top of the OCR API, the OCR-Vdm can achieve good performance in a small-scale distributed environment, even without distributed task scheduling and with straightforward data management. However, since the penalty for bad scheduling decisions is much higher in a large-scale distributed-memory environment, we also plan to include more sophisticated scheduling and data management strategies in the future. This includes exploiting the hints that the applications will be able to provide to the runtime in the next version of the OCR API as well as information available from PDL descriptors as discussed in Section 6.1. We also plan to explore the use of task implementation variants, similar to the support for different codelets for CPU/GPU-based systems as pioneered by StarPU [1, 10], to allow the runtime to automatically select CPU or OpenCL variants of a task in order to efficiently target heterogeneous parallel architectures equipped with GPUs and other types of accelerators.

Another direction of work will be on the investigation of how to effectively map higher-level programming abstractions onto the OCR runtime. In particular we are working on using the OCR within the PEPPHER programmability and performance portability framework [2, 3, 13].

Given the increased performance variability of future parallel architectures we believe that automatic performance tuning methods will be an essential part of any software stack for parallel systems and we plan to extend our previous work on autotuning [6, 9, 16] also to OCR and OCR-based applications.

8. REFERENCES

[1] 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; Euro-Par 2009*, 23:187–198, 2011.

[2] S. Benkner, E. Bajrovic, E. Marth, M. Sandrieser, R. Namyst, and S. Thibault. High-level support for pipeline parallelism on many-core architectures. In C. Kaklamanis, T. Papatheodorou, and P. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 614–625. Springer Berlin Heidelberg, 2012.

[3] S. Benkner, S. Pllana, J. Traff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. Peppher: Efficient and productive usage of hybrid computing systems. *Micro, IEEE*, 31(5):28 –41, sept.-oct. 2011.

[4] J. Bueno, J. Planas, A. Duran, R. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *Parallel Distributed Processing Symposium (IPDPS 2012)*, 2012.

[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. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 280–289, 2013.

[6] J. Dokulil and S. Benkner. Automatic tuning of a parallel pattern library for heterogeneous systems with Intel Xeon Phi. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, pages 42–49, Aug 2014.

[7] J. Dokulil and S. Benkner. OCR extensions - local identifiers, labeled GUIDs, file IO, and data block partitioning. *CoRR*, abs/1509.03161, 2015. `http://arxiv.org/abs/1509.03161`.

[8] J. Dokulil and S. Benkner. Retargeting of the Open Community Runtime to Intel Xeon Phi. In *International Conference On Computational Science, ICCS 2015*, pages 1453–1462. Procedia Computer Science, 2015.

[9] J. Filipovic and S. Benkner. Opencl kernel fusion for gpu, xeon phi and cpu. In *Proceedings of IEEE International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2015.

[10] A. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. Composing multiple StarPU applications over heterogeneous machines: A supervised approach. *The International Journal of High Performance Computing Applications*, 28:285 – 300, Feb. 2014.

[11] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX - a task based programming model in a global address space. In *PGAS 2014: The 8th International Conference on Partitioned Global Address Space Programming Models*, 2014.

[12] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proc. of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, USA, 1993. ACM.

[13] C. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J. Traff, and S. Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1403–1408, March 2012.

[14] A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(04):309–322, 2007.

[15] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar. *OCR The Open Community Runtime Interface, version 1.0.0*, June 2015.

[16] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. In P. Manninen and P. Öster, editors, *Applied Parallel and Scientific Computing*, volume 7782 of *Lecture Notes in Computer Science*, pages 328–342. Springer Berlin Heidelberg, 2013.

[17] M. Sandrieser, S. Benkner, and S. Pllana. Improving programmability of heterogeneous many-core systems via explicit platform descriptions. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 17–24, New York, NY, USA, 2011. ACM.

[18] M. Sandrieser, S. Benkner, and S. Pllana. Using explicit platform descriptions to support programming of heterogeneous many-core systems. *Parallel Computing*, 38(1-2):52–65, 2012.