The Open Community Runtime: A Runtime System for Extreme Scale Computing

Timothy G. Mattson*, Romain Cledat*, Vincent Cavé*, Vivek Sarkar†, Zoran Budimlić†, Sanjay Chatterjee†, Josh Fryman*, Ivan Ganev*, Robin Knauerhase*, Min Lee*, Benoît Meister‡, Brian Nickerson*, Nick Pepperling*, Bala Seshasayee*, Sagnak Tasirlar¶, Justin Teller§, and Nick Vrvilo†

*Intel Corporation, Hillsboro OR, USA †Rice University, Houston TX, USA
‡Reservoir Labs, New York NY, USA §Facebook, Menlo Park CA, USA
¶Two Sigma Investments, LP

***Abstract*—The Open Community Runtime (OCR) is a new runtime system designed to meet the needs of extreme-scale computing. While there is growing support for the idea that future execution models will be based on dynamic tasks, there is little agreement on what else should be included. OCR minimally adds events for synchronization and relocatable data-blocks for data management to form a complete system that supports a wide range of higher-level programming models. This paper lays out the fundamental concepts behind OCR and compares OCR performance to that from MPI for two simple benchmarks. OCR has been developed within an open community model with features supporting flexible algorithm expression weighed against the expected realities of extreme-scale computing: power-constrained execution, aggressive growth in the number of compute resources, deepening memory hierarchies and a low mean-time between failures.**

I. Introduction

Programming models for high performance computing evolve slowly. Researchers develop programming models by the thousands, but very few are actually used. On rare occasions, new programming models emerge from the research community and become adopted by a wider community of users. It may appear that this happens because programmers are attracted to the productivity benefits or other features of a new system. While productivity is indeed an important factor in deciding between two systems which solve a similar problem, the adoption of a new system is primarily driven by an external change. MPI [15] emerged because the "attack of the killer micros" made distributed memory MIMD computers almost ubiquitous in supercomputing. OpenMP [13] emerged because SMP systems became common with the emergence of commercial off the shelf chip-sets that supported multi-socket systems. Multi-core chips came later and further solidified the importance of OpenMP. Programmable rendering pipelines in GPUs created the GPGPU movement and put CUDA [11] on the map.

The pattern is clear: in each case it was not the features of a programming model that drove its adoption by the applications community but rather the need to support a changing hardware reality that turned research parallel programming systems into new mainstream parallel programming models.

With this perspective in mind, consider the state of exascale computing. The details vary depending on which vendor is making the pitch, but the high-level features of the hardware for these extreme scale systems are clear; they will most likely be characterized by:

- Power optimized designs based on large numbers of low power cores. Systems with tens or even hundreds of millions of cores would not be surprising.
- The cores making up the system will be packaged into processors and organized into deep hierarchies. Instruction sets are likely to vary across the system. Even if the cores all utilize the same instruction set, the variation in frequency between cores will basically make the system one in which programmers need to manage heterogeneity.
- The memory hierarchy will deepen significantly and contain multiple types of memory, not just DRAM with a cache hierarchy. There will be persistent memory, in package DRAM, scratchpads, node level DRAM, and additional memory capabilities exposed across the system.
- With so many components, the mean time between failure of some component in the system will be short relative to the runtime of a computation. With such a large, spatially distributed system, a checkpoint stored into a central store will not be possible. Hence, a computation must be able to recover and proceed should a component of the system fail.

These last three points suggest that runtime systems for extreme scale computers will need to decouple computational work and data from the system resources used to support them. The work in a computation can be decomposed into a large number of asynchronous tasks that can migrate around faults and distribute themselves about the processing elements of a system to effectively balance the load. Likewise, to support task migration and resilience models, the data within a computation should be represented as relocatable objects virtualized from the supporting memory. A common instantiation of these concepts can be found in Asynchronous Many Task (AMT) models.

We submit that the changes suggested by exascale computers are significant enough to drive application developers to consider alternatives to the dominant programming models in use today (OpenMP and MPI). The transition from terascale to

petascale was accomplished without changing the underlying programming models. With the shift to exascale computers, however, it is likely that programmers will have no choice but to adopt new programming models based on AMT runtime systems.

The Open Community Runtime, or OCR, is a new AMT system designed to address the challenges of exascale systems. In this short paper, we introduce OCR, describe the motivation behind it, and present results with a few simple benchmarks. We do not have space, however, to present all the features of the runtime or describe how our reference implementation of OCR is implemented.

II. The Open Community Runtime

The Open Community Runtime (OCR) is an AMT runtime system for extreme scale computers. It is a community driven project defined by a formal specification [9] providing a common foundation for independent implementations.

OCR is a *r*untime system. OCR's low-level API suggests a natural programming model, but most application programmers will prefer higher-level programming models that run on top of OCR. RStream [10], [19], CnC [2], and Habanero-C++ already target OCR and we are currently working on porting Legion [1] and OmpSS [5] to OCR. These porting efforts validate our hypothesis about the utility of AMT systems and help us refine the OCR API to more effectively support a wide range of high-level programming models.

An OCR program defines a computation as a collection of asynchronously executing tasks operating on relocatable *data-blocks*. *Events* coordinate the execution of OCR tasks; hence an OCR task is called an *Event Driven Task* or *EDT*. Logically an OCR computation is a Directed Acyclic Graph (DAG) where the nodes are OCR objects (data-blocks, events or tasks) and the edges are dependences between objects. When the dependences of an EDT are met, the EDT is guaranteed to run at some point in the future.

All persistent state in an OCR program is defined through data-blocks. In OCR, EDTs are transient objects; they run without blocking and when complete, are destroyed by the runtime system. Hence an EDT appears to other EDTs as a transaction. This allows the state of a computation to be defined by a log of which EDTs have executed on which data-blocks. This will serve as a foundation for a formal resilience model to appear in a future version of OCR.

A. OCR objects

As previously stated, an OCR program is a DAG of OCR objects, each of which can be referenced through a globally unique identifier (GUID). The edges in the DAG are explicitly created dependences between *objects*. Dependences are defined in terms of *slots* on an object with input or *pre-slots* and output or *post-slots*. A *dependence* is a *link* between the *post-slot* of one object and a *pre-slot* of a different object. This defines a producer-consumer relationship between objects.

A post-slot can be connected to multiple pre-slots but a pre-slot can only be connected to one post-slot; thereby defining a well defined "flow" for data-blocks along dependence links. A pre-slot is said to be *satisfied* if it is connected to a post-slot and if that post-slot is also satisfied. In other words, the satisfaction of a post-slot will *trigger* the satisfaction of all pre-slots connected to it. The exact significance of pre- and post- slots is different for each object and is explained in the following sections.

1) Event Driven Tasks (EDTs): An EDT is the basic unit of computation in OCR. EDTs encapsulate both the user code (the "useful" application computation) as well as the code to setup the application's DAG. An EDT can have 0 to n pre-slots and a single post-slot associated with an event. The satisfaction of an EDT's post-slot indicates its completion.

An EDT's pre-slots determine both the lower-bound for *when* it can execute and *what*, in terms of data-blocks, it has access to. An EDT will only become *runnable* once all of its pre-slots have been satisfied and, when it does run, can only access data-blocks that have either been passed in through its pre-slots or that it creates itself during the course of its execution. There are several "rules" for an EDT to be compliant with the OCR programming model:

1) An EDT must be re-entrant. This precludes the use of static or global data.
2) An EDT may only read or write to data-blocks it has access to.
3) An EDT cannot perform any synchronization operation that might cause it to block (e.g. a lock) outside of the synchronization operations provided by the OCR API.

The importance of these rules will become clear when we discuss the OCR execution model in Section II-B.

2) Data-blocks (DBs): In OCR, a data-block is a relocatable block of memory. A data-block has no pre-slot and a single post-slot which is always in a satisfied state. Hence, once created data is always considered to be *available*. Data-blocks cannot be used until they are *acquired* which, in effect, translates the GUID into a pointer usable by the EDT. In OCR, this can only happen when an EDT starts its execution or at the creation of the data-block. Note that the "availability" of a data-block does not mean "all EDTs can use it at any time". Since the runtime is responsible for scheduling EDTs, it can place data-blocks and schedule EDTs appropriately to make sure that the data-block is readable by the EDT when acquired. When a data-block is no longer being used, it should be *released*. This happens either explicitly through an API call or implicitly at the end of the acquiring EDT. This acquire/release semantic is key to the OCR memory model (see Section II-D).

Data-blocks are the only way for EDTs to share data. Hence, the state of a computation is defined by the collection of released data-blocks and a log of EDT execution; a fact we will exploit in future resilience models. Additional details for how EDTs communicate data are detailed in Section II-B.

3) Events: Events are OCR's synchronization mechanism. An event has at least one pre-slot and exactly one post-slot. An event can be understood in terms of two functions.

1) *Trigger function*: A function that determines when the post-slot of the event becomes satisfied based on the state

of its pre-slots.

2) *Output function*: a function that determines the data, if any, that will be passed along the post-slot of the event once it becomes satisfied.

The simplest form of event has one pre-slot, a trigger function that satisfies the event's post-slot upon satisfaction of the event's single pre-slot, and an output function that transfers whatever came in on the event's pre-slot. Other types of events are defined in the specification but are beyond the scope of this paper.

B. Execution model

The work of an OCR program is defined in terms of a collection of tasks organized into a directed acyclic graph (DAG) [16], [17], [20]. Task execution is controlled and managed through events; a task can only execute once all of its pre-slots, often connected to events, are satisfied.

All API functions described in this section are illustrated in Section II-C.

1) Execution platform: An OCR program executes on an abstract machine called the *OCR platform* which consists of:

- A collection of network connected nodes.
- Each node consists of one or more processing elements each of which may have its own private memory. A node may also contain memory regions that are shared by the processing elements of the node.
- Workers that run on the processing elements to execute EDTs.
- A globally accessible shared name space of OCR objects each denoted by a globally unique ID (GUID).

OCR is designed to be portable and scalable, hence the OCR platform places minimal constraints on the physical hardware. In particular, OCR does not assume a globally shared address space with a cache-coherent memory hierarchy. It only requires a a shared globally unique name space.

The execution of an OCR program logically starts as a single EDT called *mainEDT()* executing on a single processing element. The DAG corresponding to the program is dynamically constructed and executes until a call is made to one of the termination APIs; either *ocrShutdown()* or *ocrAbort()*. Any EDT may call a termination API. The termination of the program, however, will only be precise if the calling EDT is the last running or runnable EDT in the program. OCR does not attempt to detect program termination, so an OCR program will hang indefinitely if one of the termination APIs is not called.

2) Dependences: OCR objects are linked together using *dependences*. There are two main mechanisms for defining dependences: **a)** the *ocrAddDependence()* API call which simply adds a link between the post-slot of the source object and the designated pre-slot of the destination object, and **b)** the *ocrEventSatisfy()* API call which satisfies the pre-slot of an event with the specified data-block.

3) EDT execution states: To understand the execution model of OCR, consider the discrete states and transitions of an EDT as defined in Figure 1. When an EDT is created, it is assigned a GUID and is said to be *Available*. An *Available* EDT knows the code that it will execute as well as the number of pre-slots that it has. The GUID of an *Available* EDT is also valid to be used in calls such as *ocrAddDependence*. The EDT is not, however, fully defined in terms of its place in the DAG until all of the EDT's pre-slots are "connected" at which point it becomes *Resolved*. Note that the transition from *Available* to *Resolved* is not called out as a named transition because it is not generally possible for the system to set a distinct time-stamp corresponding to when the transition occurred. In this case, the transition is un-named because dependences may be defined dynamically up until the EDT *Launch* transition. At this point the EDT is *Runnable*.

Once the EDT is *runnable*, it will execute at some point during the normal execution of the OCR program. Prior to starting execution, all data-blocks passed on the pre-slots of the EDT will be acquired and the EDT becomes *Ready*. The *Ready* EDT will be scheduled by the OCR scheduler and *Start* executing to become a *Running* EDT.

Once the EDT transitions to the *Running* state, it is guaranteed to eventually *Finish* and transition to the *End* state. At some later point, the EDT will release all acquired data-blocks not explicitly released by the user and enter the *Released* state. At this point, all changes made by the EDT to any of its acquired data-blocks will be available for use by other OCR objects (see Section II-D). Later, the EDT will *Trigger* the event associated with it thereby becoming a *Triggered* EDT and potentially satisfying events waiting on its completion. At some later point, the system will *Clean-up* the resources used by the EDT (potentially recycling its GUID) and the EDT is destroyed.

C. Illustrative example

In this section, we describe some of the functions from the OCR API and how they are used to implement a "simplistic" Fibonacci example. The basic Fibonacci computation computes $F(n)$ using the recurrsion $\forall n \geq 2 : F(n) = F(n-1) + F(n-2)$ with initial values $F(0) = F(1) = 1$. In this implementation, we use three EDTs to compute $F(n)$: one to compute $F(n-1)$, one to compute $F(n-2)$ and one to perform the summation.

Due to space limitations, we only cover the main points in this section. The full program is available at [6]. This section describes the low-level OCR API and the low-level programming model naturally exposed by OCR.

a) Creating tasks: A task is represented by an EDT in OCR. An EDT is submitted for execution as follows:

```
typedef ocrGuid_t (*ocrEdt_t)(u32 paramc,
                  u64* paramv, u32 depc,
                  ocrEdtDep_t depv[])
```

It takes $paramc$ parameters ($paramv$) known at creation time and $depc$ input dependences, some of which may be associated with data-blocks while others may be pure control dependences. It returns the identifier (a GUID) of a data-block that is eventually passed to the successors of the EDT.

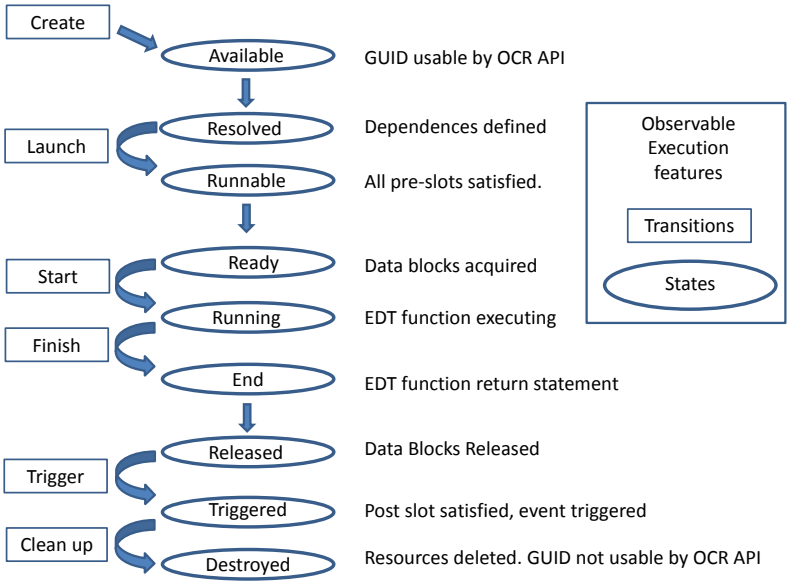

Fig. 1. Observable execution features

Creating an EDT is a two-step process: **a)** a template is created to describe the EDT type and define the function that the EDT will execute ($funcPtr$), then **b)** an instance of the EDT from the indicated template ($templateGuid$) is created for each task. The OCR functions involved are:

```
u8 ocrEdtTemplateCreate(ocrGuid_t *out,
      ocrEdt_t funcPtr, u32 paramc,
      u32 depc);

u8 ocrEdtCreate(ocrGuid_t *edtGuid,
      ocrGuid_t templateGuid,
      u32 paramc, u32 depc,
      ocrGuid_t *depv,
      u16 properties, ocrHint_t *hint,
      ocrGuid_t *outputEvent);
```

Like all OCR APIs, an error code is returned and the output values (such as the GUID for the created objects) are returned through the first pointer parameter.

In the Fibonacci example, *mainEdt*, which is the starting point of the computation, will create an EDT that will execute at the end of the program and print out the result as well as an EDT to compute $F(n)$. That EDT will, in turn, create three other EDTs: one to compute $F(n-1)$ and $F(n-2)$ and one to perform the sum.

b) Creating data-blocks: Creating a data-block is very similar to allocating a chunk of memory; the following API creates a block of memory of length len and returns a pointer ($addr$) and identifier (out):

```
u8 ocrDbCreate(ocrGuid_t *out, void** addr,
        u64 len, u16 flags, ocrHint_t *hint,
        ocrInDbAllocator_t alloc)
```

In the Fibonacci example, a data-block is created for each input value as well as the resulting output value.

c) Creating the task-graph: Events are OCR's mechanism for synchronizing EDTs. Events are created using the following API:

```
u8 ocrEventCreate(ocrGuid_t *evtGuid,
      ocrEventTypes_t type, u16 properties)
```

In our example, we create events to:

- Link the completion of the computation of $F(n-1)$ to the EDT that performs the summation.
- Link the completion of the computation of $(n-2)$ to the EDT that performs the summation.
- Link the summation EDT to the consumer of that value (typically $F(n+1)$).

These dependences are added using the following function call:

```
u8 ocrAddDependence(ocrGuid_t source,
      ocrGuid_t destination, u32 slot,
      u16 properties)
```

d) Exchanging data: To exchange data between EDTs, an EDT must satisfy the event on its post-slot and associate that event with the indicated data-block:

```
u8 ocrEventSatisfy(ocrGuid_t event,
                   ocrGuid_t db)
```

This function satisfies an event ($event$) and possibly triggers another EDT while passing data (db) along its dependence vector.

D. Memory Model

1) Definitions: To define the memory model for OCR, we define two relations:

- **Sequenced-before** is the relation between two ordered operations in an EDT as defined by the C programming language. This is commonly referred to as "program order"
- **Synchronized-with** is the relation between two points in an OCR program. The only synchronized-with relation in OCR is obtained through the use of events: the satisfaction of a pre-slot of an event may synchronize with the triggering and satisfaction of its post-slot. For simple events, this means that the *ocrEventSatisfy()* call on an event X synchronizes with the satisfaction of the pre-slots of any object (EDT or event) connected to the post-slot of X.

These two relations allow us to define a **happens-before** relationship: if A is sequenced before B in EDT1 and C is sequenced before D in EDT2 and B is synchronized with C then A happens before D.

2) Acquire/release model: OCR provides a relatively simple memory model: before an EDT can read or write a data-block, it must first *acquire* the data-block. This is not an exclusive relationship by which we mean it is possible for multiple EDTs to acquire the same data-block at the same time. When an EDT has finished with a data-block and it is ready to expose any modifications to the data-block to other EDTs, it must *release* that data-block.

To precisely define the **happens-before** relationships, the runtime and the user must collaborate. The OCR runtime ensures the following property:

> When an EDT releases a data-block D, either explicitly or implicitly when the EDT ends, all loads and stores to that data-block complete before the data-block is released and the release appears to complete before the release call returns. Note that this does not mean that the release necessarily fully completes before the call returns (as this could be a long latency operation) but rather that any other OCR calls will only execute on a state that reflects that the release has completed.

The user must adhere to the following rule:

> Before an EDT calls a function to satisfy an event, any data-block potentially exposed to other EDTs by that event satisfaction must be released prior to the event satisfaction. All writes to the data-blocks must also be sequenced before the release API call.

With these two properties, the user can be certain that any EDT B that happens *after* an EDT A will see the modifications that A made to a data-block D if D is available to B.

3) Data races: The OCR memory model described above defines the behavior for EDTs that have a clear happens-before relation. OCR, however, lets two or more EDTs write to a single data-block at the same time (or, more precisely, the two EDTs can issue writes in an unordered manner). This capability can lead to data races if conflicting accesses are performed on the same byte/word of the shared data block. Allowing data races in an OCR program is not an oversight but a deliberate decision to allow OCR to scale better on a wider range of parallel computers and not overly constrain hardware OCR programs run on. The rules that OCR uses to handle writes to a single data-block by multiple EDTs are straightforward and can be found in the OCR specification[9] in Section 1.5.2.

III. Experimental Results

We are aware of three independent implementations of OCR; OCR-Vx [4] from the University of Vienna, a recent implementation from the Pacific Northwest national laboratory, and our own XSOCR system. We used XSOCR for this study.

XSOCR was developed as part of the DoE's XStack project [12]. It targets a single shared-memory x86 node, clusters of x86 nodes using either MPI or Gasnet, and the FSIM simulator of a "straw-man" exascale computer. The goal of XSOCR was to explore features of OCR and to experiment with different implementation strategies; it was not optimized for any particular platform.

Details about XSOCR and its design are beyond the scope of this short paper. Our goal here is to demonstrate that working versions of OCR are available. A full analysis of performance results will be presented in a future paper.

To study the performance of OCR and compare to MPI, we considered two simple benchmarks: Stencil-2D and HPCG. Stencil-2D is a simple two dimensional, explicit stencil code. HPCG is the well known High Performance Conjugate Gradient Benchmark.

For the OCR version of the benchmarks we used a straightforward SPMD approach. For both benchmarks the domains are decomposed in data-blocks with one EDT per sub-domain responsible for an iteration of the computation. The halo exchanges are done by satisfying neighbors' events with data-blocks holding border values. Events are used as control and data dependences to schedule EDTs for the next iteration of a sub-domain.

All results reported in this paper were produced on Edison; a Cray XC30 computer at NERSC. The nodes in Edison consist of two Intel® Xeon® E5-2695 Processors (Ivy Bridge) with 64 Gbytes of memory. Each processor has 12 cores for a total of 24 physical cores per node. The nodes are connected by a Cray Aries Dragonfly network.

We used OCR v1.1 and Intel MPI 5.2. OCR benchmarks were run as one process per cluster node with as many worker threads as cores whereas MPI benchmarks were run as one process per core. Note that OCR dedicates one worker to manage communications so there is effectively one less computation worker. The stencil-2D benchmark uses all 24 cores of a cluster node and weak scaling is performed from 1 to 1024 nodes (23 to 23552 computation workers). The initial input size is an 8640x8640 elements grid (1.1 GB).

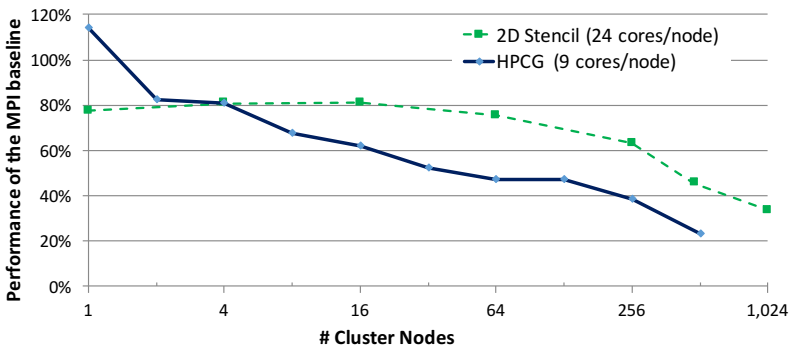

Fig. 2. Stencil-2D and HPCG performance for OCR reported as a percentage of the performance of the corresponding programs implemented in MPI. All runs were on the Edison system at NERSC. On one node, Stencil-2D ran at 42 GFlops for MPI and 32 GFLOPS for OCR. On 1024 nodes, Stencil-2D ran at 4 PFLOPS with MPI and 1.3 PFLOPS with OCR.

The HPCG benchmark uses 9 cores out of the 24 cores of a cluster node because of memory bandwidth saturation. Weak scaling is performed from 1 to 512 cluster nodes (8 to 4096 computation workers). The initial input size is 64x64x64 where each worker owns a 2x2x2 chunk. Every time resources double, the next dimension size is doubled starting from the first.

Figure 2 shows weak scaling results. The OCR implementation of Stencil-2D achieves about 80% of the performance of the reference MPI implementation from 1 to 64 nodes, which, for a relatively unoptimized runtime (when compared to the years of work that have gone into optimizing MPI runtimes), is very promising. A modest performance decline from 64 to 256 nodes (75% to 63%), and a sharper decline on to 33%.

The OCR implementation of HPCG shows a 14% improvement on a single node with respect to MPI. This is due to the fact OCR operates as a single process with 8 computation workers whereas MPI is ran as 8 individual processes. For distributed runs of HPCG, the OCR implementation achieves 80% of the MPI performances up to 4 nodes and degrades as node count increases to 20% for 512 nodes. The reason for the poor scaling of the OCR program is the excessive overhead in the OCR reduction library.

IV. Related Work

The design of OCR was influenced by the codelet [21] execution model at University of Delaware, and the Habanero execution model at Rice University, which in turn was influenced by early work on X10 [3] at IBM. All of these projects built on the idea of lightweight asynchronous tasks popularized by the MIT Cilk [14] project. The Habanero project introduced data-driven tasks (DDTs) and data-driven futures (DDFs), which formed the conceptual basis for EDTs and events/data-blocks in OCR. A major motivation for Habanero DDTs and OCR EDTs was to expand the scope of asynchronous task parallelism beyond manycore processors to also encompass heterogeneous accelerator-based parallelism and distributed-memory parallelism, both of which are not addressed by fork-join task parallel models such as Cilk. Further, the implementation of a scalable work-stealing task scheduler with the help-first scheduling policy in Habanero-C directly led to the initial implementations of task scheduling in OCR.

HPX [7] is another task-based runtime and programming model aimed at future ExaScale systems. HPX implements an active global address space, where globally-addressable objects can be tracked by a unique global ID as they are migrated throughout the system.

Charm++ [8] takes an object-oriented approach to building ExaScale software. Charm++ programs are decomposed into distributed objects, called *chars*, which are distributed throughout a system by the runtime. Messages are passed throughout the system via method calls on remote *char* objects.

Realm [18] is a fully asynchronous, event-based runtime for task-based computations. All runtime actions in Realm are non-blocking, building on a lightweight event mechanism for dependence management. Realm implements a concept of *physical regions* for shared global data, which provides type information for blocks of data that may be migrated to remote locations by the runtime. The additional type information provided for *physical regions* allows Realm to combine compute operations, such as reductions, with data movement within the runtime.

V. Conclusion

OCR is a new runtime system designed to address the challenges of extreme-scale computing. The chief contributions of this paper were to introduce OCR, its execution and memory models, and to provide some early performance data. In particular, we set out to show how the fundamental concepts in OCR (tasks, events, and data-blocks) provide the virtualization needed to support the resilience and the dynamic load balancing required by future exascale systems.

Acknowledgements and Disclaimers

This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717.

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[2] Z. Budimlic, M. Burke, V. Cave, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar. Concurrent collections. *Scientific Programming*, 18(3):203 – 217, 2010.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[4] J. Dokulil, M. Sandrieser, and S. Benkner. Ocr-vx - an alternative implementation of the open community runtime. In *International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15. Austin, Texas, November 2015*, November 2015.

[5] A. Duran, E. Ayguade, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173 – 193, 2011.

[6] Intel Corporation. Fibonacci code in OCR. https://goo.gl/9Ojcb8.

[7] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

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

[9] T. Mattson, R. Cledat, Z. Budimlić, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar. Ocr, the open community runtime interface. Technical report, Intel Corporation and Rice University, 2015. OCR Specification 1.0.1.

[10] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin. R-stream compiler. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer, 2011.

[11] Nvidia. NVidia CUDA Programming Guide version 7.5. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2015.

[12] OCR Core Team. XStack OCR Wiki. https://xstack.exascale-tech.com/wiki.

[13] OpenMP Application Program Interface, version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[14] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[15] A. Skjellum, E. Lusk, and W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[16] S. Taşırlar and V. Sarkar. Data-Driven Tasks and their Implementation. In *ICPP'11: Proceedings of the International Conference on Parallel Processing*, Sep 2011.

[17] S. Tasirlar. Scheduling Macro-Dataflow Programs on Task-Parallel Runtime Systems, Apr 2011.

[18] S. Treichler, M. Bauer, and A. Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 263–276, New York, NY, USA, 2014. ACM.

[19] N. Vasilache, M. M. Baskaran, T. Henretty, B. Meister, H. Langston, S. Tavarageri, and R. Lethin. A tale of three runtimes. *CoRR*, abs/1409.1914, 2014.

[20] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.

[21] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.