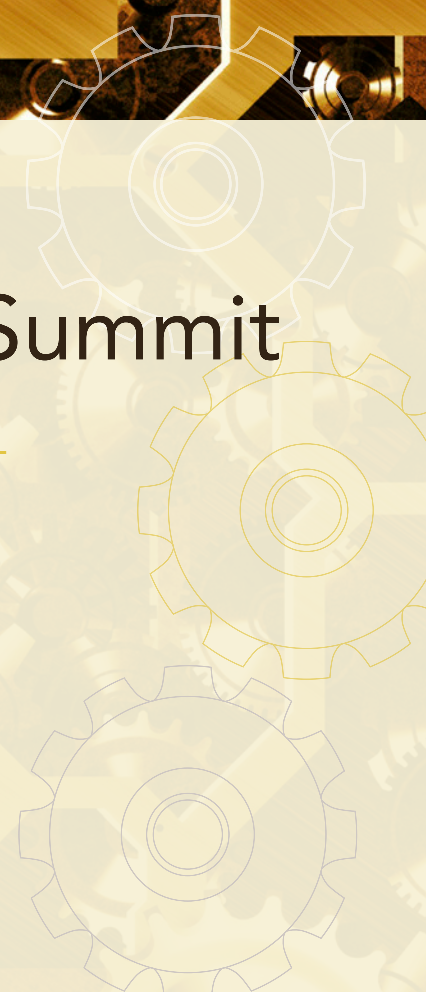
RUNTIME SYSTEMS REPORT

2014 Runtime Systems Summit



EDITORS: Vivek Sarkar, Zoran Budimlic, and Milind Kulkarni

CONTRIBUTORS: Ron Brightwell, Andrew A. Chien, Maya Gokhale, Laxmikant Kale, Milind Kulkarni, Rich Lethin, Wilfred Pinfold, Vijay Saraswat, Vivek Sarkar, Thomas Sterling, and Katherine Yelick

DOE ASCR REVIEWER: Sonia R. Sachs

September 19, 2016

U.S. DEPARTMENT OF ENERGY | Office of Science

Sponsored by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR)

17-CS-3270

Acknowledgments

The authors would like to acknowledge the contributions of all participants in the Runtime Summit held in Washington D.C. on April 9, 2014, as well as input from participants in the DOE X-Stack and OS/R programs.

Executive Summary

This report summarizes runtime system challenges for exascale computing, that follow from the fundamental challenges for exascale systems that have been well studied in past reports, e.g., [6, 33, 34, 32, 24]. Some of the key exascale challenges that pertain to runtime systems include parallelism, energy efficiency, memory hierarchies, data movement, heterogeneous processors and memories, resilience, performance variability, dynamic resource allocation, performance portability, and interoperability with legacy code.

In addition to summarizing these challenges, the report also outlines different approaches to addressing these significant challenges that have been pursued by research projects in the DOE-sponsored X-Stack and OS/R programs. Since there is often confusion as to what exactly the term "runtime system" refers to in the software stack, we include a section on taxonomy to clarify the terminology used by participants in these research projects. In addition, we include a section on deployment opportunities for vendors and government labs to build on the research results from these projects. Finally, this report is also intended to provide a framework for discussing future research and development investments for exascale runtime systems, and for clarifying the role of runtime systems in exascale software.

Chapter 1

Introduction

1.1 Exascale context

The primary context for this report lies in challenges for exascale systems that have been described in depth in past reports and studies [6, 33, 34, 32, 24]. One of the trends for exascale systems is that the bigger disruption for runtime systems will occur at the intra-node level rather than the inter-node level. This is because the degree of intra-node parallelism needs to increase by two to three orders of magnitude relative to today's high-end systems on the path to exascale, while the degree of inter-node parallelism only needs to increase by at most one order of magnitude. Further, the impact of increasing heterogeneity at the processor and memory levels will be felt more acutely at the intra-node than the inter-node level, as will the impact of energy constraints. Taken together, these challenges point to the need for a clean sheet approach to intra-node runtime systems, which in turn will have a significant impact on existing inter-node communication runtimes[1] (such as MPI or PGAS communication runtimes) due to the hybridization challenge: *any successful inter-node runtime system must be able to integrate well with new intra-node runtimes for exascale systems.*

1.2 X-Stack context

The Department of Energy's exascale software stack program (X-Stack) is exploring novel ideas for programming systems for exascale machines in the 2023 timeframe. Driven by power constraints and diminishing returns on traditional uniprocessor scaling, the architectural landscape is undergoing a radical transformation relative to the networks of single core and simple multicore systems of the past decade. The goal of the X-Stack program is to enable scientific discovery and the solution of mission-critical problems on future exascale systems. This implies both performance and productivity goals, driven by the needs of the application community on the one hand and constrained by the available hardware approaches on the other. There is uncertainty on both sides, e.g., what future hardware will look like, how future applications and algorithms will be designed, and how they will be implemented on future hardware. This uncertainty makes the design of an effective, high performance, portable programming system especially challenging. The X-Stack program is exploring a number of approaches in languages, compilers, runtime systems and tools to express, manage and automate various aspects of parallelism, locality, communication, scheduling, and variability. The purpose of this report is to describe the role of the runtime system in this research.

1.3 OS/R context

The Department of Energy's operating system and runtime (OS/R) program is also exploring novel approaches to supporting advanced runtime systems, but doing so from a bottom-up view of the software stack rather than the top-down view in the X-Stack program. The operating system (OS) and the runtime system must coordinate much more closely than before on the allocation and management of resources both at the

[1] For brevity, we will use the terms "runtime system" and "runtime" interchangeably in this report.

basic component level as well as at the system level. Traditional OS/R functions and interfaces provide limited support for the dynamic discovery and adaptation capabilities that future runtime systems require. Exploration and advancements will be needed in areas such as methods and mechanisms for efficient communication between the OS and runtime system, approaches for adaptive resource management policies, new techniques for isolation and sharing of resources, and efficient ways of handling dynamic discovery and configuration of hardware state. Future operating systems for exascale systems will play a critical role in enabling runtime systems to map the applications on to exascale machines.

1.4 Vision

Runtime systems are expected to be responsible for managing the inherent complexity in future extreme-scale computing systems, and efficiently managing hardware resources to support application goals. The current approach to managing complexity in extreme-scale systems is to provide generalized abstractions and machine models that allow algorithm designers and application developers to create code that will work reasonably well on a broad spectrum of computing systems. Compilers, libraries, runtime systems, and operating systems work within the constraints of these abstractions to map the application to the underlying hardware as efficiently as possible. Performance analysis tools identify potential opportunities for the application or library developer to employ techniques that allow the compiler or the system software to make better use of hardware resources. Recent progress on more dynamic runtime systems has been the result of raising the level of the abstraction presented to the application developer, thereby giving the runtime system more opportunities to better exploit parallelism exposed by less constrained models (as exemplified by deferred execution models with asynchronous tasks). In this way, advances in runtime systems can support the advances in both programmability and performance portability that are necessary for exascale systems.

Runtimes for future extreme-scale systems will continue to make improvements that provide better, more complete abstractions and models beyond the node level, potentially increasing the ability of the runtime system to not only map the application to the machine, but also to discover methods and techniques that allow mapping the machine to the application. Today's runtime systems still require the application or library developer to explicitly define important aspects of the problem, such as granularity of computation or number of nodes required, constraining the ability to adapt to changes in resources; future runtime systems should go beyond these constraints.

Dynamic discovery is a critical capability necessary to map the machine to the application efficiently and effectively. The runtime will need the ability to determine the goals of an application and discover the best way to use resources to meet those goals. The runtime will have to discover how an application behaves in response to its decisions, discover how the hardware resources are performing, and be responsible for providing performance portability across various hardware platforms and configurations.

A critical capability that future runtime systems will need to employ is the accumulation and use of knowledge about the application. Application characteristics, such as granularity of computation or communication, will be dynamically tuned by the runtime system for the particular hardware resources available during execution. Future runtime systems will be responsible for activities like load balancing, discovery of parallelism, and even the possibility of runtime recompilation to specialize critical kernels in the application.

The dynamic nature of future systems due to the heterogeneity and reliability challenges that exist across all memory, compute, and communication resources in the system, and the constantly changing cost model of these resources, motivate the need for the runtime system to be able to dynamically discover the progress of the application and the state of local and global resources. Future runtime systems must be able to create knowledge about application and the system, make informed decisions about potential optimizations, and act on these decisions at appropriate timescales. The goal is not only to improve the scalability and efficiency of the application and the effectiveness of the system, but also to improve the productivity of application and library developers by presenting models, abstractions, and interfaces that enable the runtime system to manage complexity and bear more of the burden of achieving efficient mappings between the hardware and the application.

Realizing this vision, even for current systems, is a daunting task that requires the exploration of fundamentally new and more holistic approaches that are informed, but not encumbered, by current methods. The expectation is that the knowledge gained through more forward-looking research will continue to inform

and accelerate the path of incremental advancements, at least until the volume of the breakthroughs causes a fundamental shift to more promising alternatives. Realizing this vision cannot be done by research in runtime systems alone, as there are strong connections and dependencies on other critical aspects of application development, system software, code generation and optimization, operating systems, and architectures.

1.5 Overview

The rest of the report is organized as follows. Chapter 2 summarizes the major technical challenges that will need to be addressed by exascale runtime systems. Chapter 3 defines the terminology used by participants in the X-Stack and OS/R research projects, as a preamble to Chapter 4 which includes research case studies from those projects. Chapter 5 summarizes potential deployment opportunities for vendors and government labs to build on the research results from these projects, and Chapter 6 contains our conclusions. Appendix A contains an initial runtime ontology with examples from current runtime systems.

Chapter 2

Technical Challenges

In this chapter, we elaborate on specific challenges for exascale runtime systems that follow from the overall challenges for exascale systems that have been described in past reports and studies [6, 33, 34, 32, 24]. The energy drivers for exascale are leading to architectural changes that will necessitate changes in the way software is written for the machines. There are also micro-architectural changes that, while they may not change the programming interface, could dramatically change the relative performance costs within the machine and therefore also require new algorithms and/or software models. The Abstract Machine Models (AMM) report [3] summarizes the most likely changes foreseeable at exascale, and points out that some of these may be ignored by software, while others are abstracted in a way that hides them from higher level software, and still others may be exposed directly to application level software. Thus, features of the hardware may be hidden by the programming model implementation, compiler, and runtime; may be abstracted into higher level programming concepts; or passed through, as is, for programmers to manage. The runtime can similarly choose to expose or hide system features from the application programmer; in both cases, the runtime is the level of the software stack where the "rubber meets the road" from the viewpoint of mapping the application on to the hardware. The AMM report describes a canonical exascale node, as shown in Figure 2.1. As with past high-end computing systems, an exascale system will be built as a network of computing nodes; however, unlike past systems, there will be disruptive changes at the intra-node level, combined with major advances at the interconnect level, to address the constraints of building exascale systems with reasonable power budgets by placing additional burdens on the software.

The anticipated technical challenges for features that the runtime needs to expose or hide from the application programme are summarized in the following sections.

2.1 Parallelism

HPC software and runtime systems were originally focused on horizontal scaling for parallelism, when clusters were built using uniprocessor nodes. This has recently evolved to include vertical scaling with additional degrees of multicore parallelism within a node. Since the degree of parallelism has to increase by more than thousand-fold when going from petascale to exascale systems, the scalability requirements for exascale system software and runtime systems will be much higher than in previous generations of HPC systems. Even small sequential section of code in an application or runtime system can lead to significant performance bottlenecks for parallel execution (c.f. Amdahl's Law). Thus, commonly used sequential idioms will need to be identified and eliminated from every level of the software stack [39]. A simple example is the difference between specifying a summation as a sequential iteration vs. (say) a Fortran 90 SUM intrinsic for arrays.

2.2 Energy

Exascale systems will need to be built using more energy-efficient processor, memory and interconnect components than today's systems, and many of the proposed hardware techniques for improving energy efficiency will have significant impact on HPC software, e.g., due to increased heterogeneity, performance

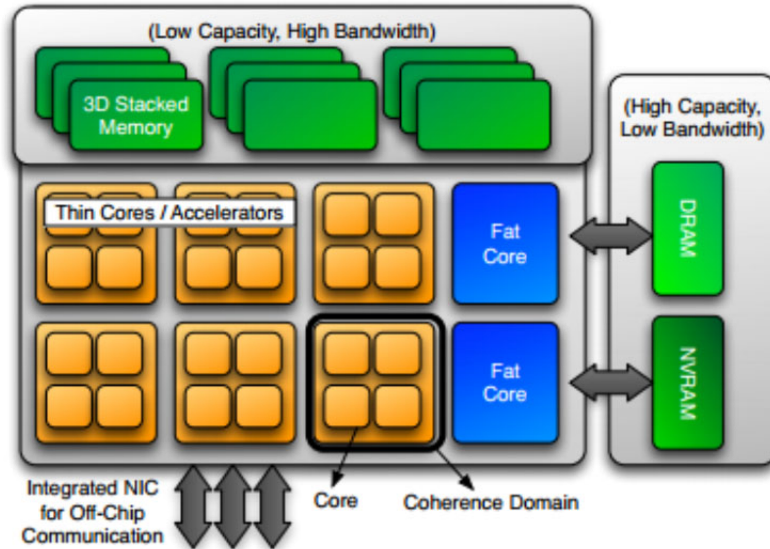


Figure 2.1: Abstract Node Architecture for Exascale

variability, and the need for reduced data movement. The runtime will need to play a major role in managing the use of power, for example by turning off and on subsets of processor cores, memories, and interconnects. Specialized cores (implemented as “dark silicon”) also offer significant improvements in energy efficiency in certain situations, and the runtime will need to play a central role in mapping computation on those cores as well. For future exascale systems, the largest contributor to energy consumption will be in data movement, which further motivates the role of the runtime in coordinating memory, locality and data movement.

2.3 Memory/Storage Hierarchy and Locality

The memory system is a significant consumer of power in modern computing systems. This is expected to drive future programming models as they focus on reducing the costs associated with moving data within a node and between nodes. Moreover, new memory technologies will be aimed at reducing the power costs associated with memory, while shifting the burden of exploiting heterogeneous memories to the software stack. The runtime will have to provide support for these new programming models and new memory technologies. Efficient exascale systems will also need to support asymmetry in node, communication, memory and I/O components. The runtime will need to allocate these asymmetric resources to irregular tasks while reducing communication and the consequent power consumed.

For example, exascale systems will need to exploit heterogeneous combinations of SRAM, DRAM, HBM (High Bandwidth Memory), NVRAM, and possibly other advanced memory technologies, to achieve the memory bandwidth and capacity targets for exascale memory and storage systems. This degree of heterogeneity in memory technologies will also have a profound impact on the role of runtime systems in supporting performance portability.

The Abstract Machine Models report [3] adds that:

> *”The need for more memory capacity and bandwidth is pushing node architectures to provide larger memories on or integrated into CPU packages. This memory can be formulated as a cache or, alternatively, can be a new level of the memory system architecture. Scratchpad memories (SPMs) have been shown to be more energy-efficient, have faster access time, and take up less area than traditional hardware cache. On-chip SPMs will become more prevalent and programmers will be able to configure the on-chip memory as cache and/or scratchpad memory, allowing initial legacy runs of an application to utilize a cache-only configuration while application variants using SPM are developed.”*

The runtime will need to assist the application in mediating this deep and heterogeneous memory hierarchy, while also providing services like software supported coherency where required.

2.4 Data Movement

Exascale systems must be able to scale efficiently to millions of nodes. Communications networks like hypercubes and meshes that follow a pre-specified mathematical topology work well in small homogeneous systems, where they simplify routing. However, more flexible communications networks that provide proportional bandwidth at multiple levels and can deal with heterogeneity will be needed for exascale.

At the node level, with concurrency scaling and (potentially) dynamic scheduling within a node, communication can become a bottleneck. The runtime must therefore manage access to limited communication resources, which includes both the physical network hardware and software support such as communication buffers. Just as we paid attention to utilization of floating point units in past decades, it will be necessary to focus on utilization of communication networks in current and future hardware. Runtime schemes that spread communication over application timesteps/iterations are therefore desirable. As network interfaces become more tightly integrated on the processor chip or silicon substrate, there are new opportunities for the runtime to use new features of the communication layer, e.g., atomic remote accesses useful for enqueueing tasks or registering events. Limited bandwidth in the internals of the network can also lead to contention and therefore resource management issues. The runtime needs to contain lightweight and flexible communication support for moving tasks and data through the system and must interact with these node level runtime services.

In summary, while interconnect details will vary from platform to platform, there is general agreement that exascale software will have to help address the need to minimize data movement due to energy limitations. Thus, the runtime capabilities discussed above will need to be locality-aware and be capable of supporting function shipping and data shipping as interchangeable alternatives. Ideally, all data movement (memory accesses, accelerator transfers, DMA transfers, inter-node messages) should be abstracted as "asynchronous data movement tasks" whose completion can trigger additional computation tasks and data movements.

2.5 Heterogeneity

The Abstract Machine Models report [3] states that:

> *"It is likely that future exascale machines will feature heterogeneous nodes composed of a collection of more than a single type of processing element. Fat cores that are found in many contemporary desktop and server processors are characterized by deep pipelines, a small number of hardware threads, a multi-level memory hierarchy, hardware to support instruction-level parallelism and other architectural features that prioritize serial performance and tolerate expensive memory accesses. The alternative type of core that we expect to see in future processors is a thin core that features a less complex design that uses less power and physical die space. By utilizing a much higher count of the thinner cores a processor will provide high performance if a greater degree of parallelism (e.g., thread-level) is available in the algorithm being executed."*

The runtime needs to consider heterogeneous architectures by dynamically assigning tasks to cores that are selected by optimizing task priority [16], data movement, core capability, core reliability, and core efficiency. In general, the runtime will need to help the applications address multiple levels of heterogeneity in exascale machines.

2.6 Resilience

Exascale systems are expected to contain sufficiently large numbers of components (hardware and software) that the aggregate mean-time-to-failure (error) will be small compared to the running time of most exascale applications. The resilience challenge includes both error detection and recovery.

The exascale runtime must be able to relocate work and data associated with a computation in response to errors arising from software or hardware. Further the runtime will need to be able to reconfigure and restart failed software components with minimal impact on application execution time, and (ideally) without requiring any user intervention.

2.7 Performance Variability

Future exascale systems will be subject to both static and dynamic forms of performance variability. Static forms include variability in node type, asymmetric communication networks and failed nodes; dynamic forms include node failures during execution or external events such as power management or thermal variability in the hardware (which in turn impact performance). The exascale runtime will need to be able to balance loads while taking both performance and power considerations into account

Achieving this balance will require introspection and feedback control. The runtime will need to be aware of available resources and their current state, understand the tradeoffs across computation, communication, memory and I/O for the jobs it has scheduled and make decisions about resource allocation that optimize system use. Exascale systems with many components will behave non-uniformly because, at any given time, nodes will be in different power states; some will have failed; some may be specialized; and there will be performance irregularities—both static and dynamic—caused by heterogeneity in storage, memory, and I/O.

Finally, the runtime for an exascale system will need to be able to schedule and re-schedule work so as to optimize performance and power use. Further, within each node, multiple data-driven work-units may be ready to execute at a time, and a scheduler in the runtime is needed to select which to execute. This control over scheduling must be leveraged by the runtime to optimize execution.

2.8 Interoperability with Legacy Code

Future runtime systems should be capable of running current legacy bulk synchronous applications efficiently, exploiting any exposed, explicit parallelism and, where possible, extracting and utilizing implicit parallelism.

Reusing constructs from existing programming systems may, at first glance, seem desirable. However, delivering these constructs at every node of the exascale system through the runtime could be onerous. Several challenges must be addressed when considering the reuse of these constructs, e.g.,

- Many of the design assumptions underlying current runtimes are fundamentally inappropriate for a large scale highly parallel system.
- The exascale runtime will need to be highly parallel, with minimal synchronization. Legacy operating and runtime systems tend to be monolithic, frequently assuming mutual exclusion for large portions of runtime code.
- High node counts put considerable pressure on overheads that may need to be borne by every node.

Chapter 3

Taxonomy

The goal of this chapter is to clarify the taxonomy used by participants in the X-Stack and OS/R research projects to define the scope of "runtime systems" in the overall software stack. It also serves as a preamble to Chapter 4 which includes research case studies from those projects.

This chapter is organized as follows. Section 3.1 describes the context for runtimes in exascale systems, specifically how the *hardware and operating system levels* provide support for runtime systems. Section 3.2 then describes key interfaces for exascale runtime systems in the form of *services*.

3.1 Runtime Systems Context

3.1.1 Hardware Support

In past decades, the HPC community focused on computer system designs that delivered the highest performance (fastest execution times) for a given technology. This approach has enabled a period of tremendous innovation and delivered extraordinary benefits. However, energy, process variability and reliability are becoming barriers to further progress on this path [7]. To reach efficient sustainable exascale performance in the face of these barriers requires a radically new and innovative approach, starting by questioning all features of the modern computing architecture that use energy, and asking if replacing them with new, more power-efficient mechanisms is possible.

Current experimental runtimes have been deployed on conventional hardware system architectures, both SMPs and scalable distributed memory systems, taking advantage of existing hardware support capabilities. However, the ultimate effectiveness of exascale runtime software will likely be realized through enhancements to future architectures that directly address key mechanisms required to reduce overheads and latencies with indirect improvements to scalability. In this section, we briefly summarize instances of hardware support for runtime software systems, both currently available and in possible future architecture extensions.

Dynamic adaptive runtimes benefit from hardware support for monitoring, control, memory, inter-node message communications, access to mass storage, as well as external I/O. While many of these forms of hardware support are also directly employed by conventional execution systems such as OpenMP and MPI, forward looking runtimes for exascale computing emphasize dynamic resource management and task scheduling—possibly with adaptive introspective control—so as to achieve significant improvement in computing efficiency with respect to conventional static practices. Such advanced methods are also intended to increase the form and reduce the granularity of parallel tasks for greater scalability, together delivering potentially dramatic improvements in performance for at least some classes of applications. However, such advantages could be offset or even eliminated if the additional overheads incurred by runtime mechanisms are too great. Hardware support can help mitigate the challenges of software overhead in runtime systems, and thereby yield the potential promise of runtime system opportunities.

While there are a number of different experimental runtimes under development, each requiring varying degrees of hardware support due in part to the distinct semantic properties of their respective underlying parallel execution models, a majority of them share various support requirements and exploit current

hardware capabilities. Possible opportunities for hardware support in current systems include (but are not limited to):

- Hardware thread execution support
- Interrupts and traps for preemptive scheduling
- Inter-process protection
- Processor status monitoring
- Intra-node memory management, page handling, and address translation
- Inter-node network message packet transfer and routing
- Network interface controller and buffering for message passing
- Processor core clock rate variation and voltage modulation
- DMA access for direct data transfer to/from network and memory
- Compound atomic operations for synchronization primitives
- Hardware queues for communicating lightweight threads/tasks among dierent functional-units/processor-cores

Future hardware support, whether incorporated individually or all together, could significantly improve efficiency and the resulting scalability. Several important opportunities exist, including:

- User-level lightweight thread creation/terminations
- Rapid context switching for user threads
- Lightweight thread preemption
- Additional hardware status data on utilization, availability, obstruction
- Support for message-driven computation and thread instantiation
- Global Address Space management (e.g., PGAS)
- Software-managed scratch pad
- NVRAM for secondary storage buffering and in-memory checkpoints
- Variable-length binary instruction sets for reduced storage and bandwidth
- In-register event synchronization
- In-memory processing for increased lightweight parallelism and higher bandwidth
- Capability based access protection support

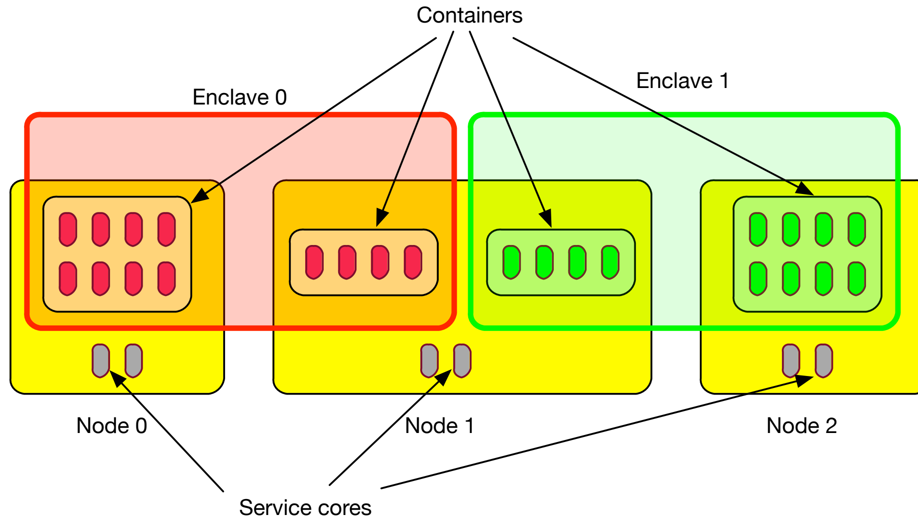


Figure 3.1: General system structure

3.1.2 Operating System Support

At exascale, it becomes more important than ever for the OS to put the application and runtime in control of resource management, providing services of protection, resource allocation, and fault isolation for jobs that span the entire machine. The exascale OS is composed of a set of hierarchical OS services: a Global OS that manages jobs at the full machine level, an enclave OS that encapsulates an application, and a node-level OS that manages containers consisting of subsets of cores and their associated resources. The general structure of the envisioned software stack is shown in Figure 1.

The global OS dynamically allocates resources to enclaves, including compute nodes or parts of those, power, transient and persistent storage and services so as to optimize throughput and turnaround requirements. The global OS maps high-level abstractions (e.g., the pipelined execution of a list of parallel components) into physical instantiations that can include the allocation of blocks of compute resources (nodes or cores) in both time and space dimensions, communication resources (network bandwidth and buffer space), and power. Workflows are managed through the global OS enclaves, with individual jobs mapped to a collection of nodes, each of which runs a node OS.

The enclave OS provides common services and functions that are common across an enclave. These functions may include the initialization and management of the communication infrastructure within the enclave, work migration and load balancing within the enclave, and recovery from failures that can be handled within the enclave.

The node level OS provides a minimal set of services to coordinate and share resources among processes executing on a compute node. The node OS may be limited to a small number of cores. It is responsible for overall node bootup, interrupt handling, process scheduling, management of heterogeneous memories and NUMA-aware resource allocation, communication management, and I/O. The node OS partitions cores (containers) and enforces isolation to minimize effects of other processes, activities across partitions. Containers may be created within the node OS, through OS-level mechanisms like Linux cgroups or virtual machine monitors, or through multiple coordinating OS kernels managing the hardware directly. This approach enables concurrent execution of jobs with possibly conflicting environment requirements to run side by side. The size of an individual container depends on application needs, but is no larger than a single compute node. Compute containers provide different capabilities. For example, an environment optimized for highly parallel HPC jobs would include predictable CPU scheduling, large, pre-faulted memory pages, and application-class-specific resource management through a customized user-space runtime library. For legacy and non-highly-parallel workloads, the container would run a more standard set of Linux services. Management of resources within one tightly coupled, parallel application is expected to be done through

application(-class)-specific user-space run-time services rather than kernel-space OS. For example, it is expected that scheduling and multiplexing of tasks among cores will be done by the runtime in user mode rather than through a kernel level scheduler.

The node-level OS also provides memory address space protection. It provides mechanisms to optimize allocation and access to heterogeneous memories including NVRAM, with non-uniform access characteristics. Inter-node communication can be handled by the node OS or can alternatively be managed by individual compute containers using virtualized network channels. I/O to node local, I/O node, and global file systems may be managed by the node OS. These individual OS services such as job launch and transmission of data through network communication or I/O are composed to support complex application workflows at the enclave level.

In addition to traditional OS services, the OS will provide services related to power management and resilience. The runtime can request the Global OS to allocate a power budget to nodes, which in turn allocates their power budget among containers. The node OS also provides access to node level power measurement sensors for the runtime to query.

For resilience, the OS will need to provide services to prevent application errors, or recover after they occurred. Such services include the provision of functions to "harden" data structures, to save data in "safe" storage, to replicate computations, etc. One key technique to improve resilience is fault-containment and hierarchical error handling. Enclaves will act as containment domains [12]: To the extent possible, errors will be detected before they have propagated outside each level in the hierarchy (container, node, enclave) will be recovered by that level, up to the level of the enclave. Errors that affect the state of an enclave are reported to the enclave; the enclave propagates error handling to the parent only if it cannot handle to error at its level. An important aspect of the exascale runtime design is to work with OS service architects on the power and fault management interactions.

3.2 Major Services

We envision a modular runtime with a common stable interface, and multiple implementations that support multiple interoperable programming environments. Each module delivers a major service in cooperation with other services, as described below.

3.2.1 Scheduling Service

We expect that a significant portion of the computational work managed by an adaptive runtime will be data-driven in nature. This may be in addition to the work explicitly scheduled by the programmer, as is currently done in MPI. To select from the work units that are ready to execute, it is necessary to have a scheduler component for each execution stream (e.g. a core or a hardware thread). The scheduler may pull work from multiple queues; some of the queues may have multiple schedulers corresponding to different execution streams as their clients. The queues may support queuing strategies ranging from simple and efficient FIFO/LIFO/DEQ to more sophisticated prioritized queues. Schedulers may hand over control to next level schedulers as needed to potentially form a stackable set of schedulers. These schedulers must coordinate to ensure that asynchronous events (such as completion of an I/O event) are handled expeditiously. Schedulers may employ different queuing strategies including FIFO, LIFO, and priority based. Examples of this component include schedulers in Argobots [37], Converse/Charm++ [18], Habanero-UPC++ [19], Qthreads [48], etc.

3.2.2 Prefetching and Asynchronous Data Transfer

The responsibility of this component is to initiate data transfer from one part of memory hierarchy to another, including scratchpad memories as well as NVRAM. Since the scheduler is in the best position to judge what will execute next, this component is closely associated with the scheduling service. In addition, the application may be able to use this component for prefetching data that it expects to use. A few research projects have also shown how asynchronous data transfers and asynchronous tasks can be "daisy chained" together using an event-driven task model as in OCR or a data-driven task as in Habanero-UPC++.

3.2.3 User-level Threads

Many of the adaptive runtimes in use today, or being proposed, incorporate user-level threads (ULTs). Argo, Qthreads, Charm++, XPRESS, FG-MPI, etc. are examples of such systems. Outside of HPC, the Boost library also includes *contexts* to support similar functionality. A common specification of the basic mechanisms for the ULTs should be standardized in consultation with these projects. Separating mechanisms from policies and only standardizing mechanisms is recommended.

Adaptive runtimes will need to migrate ULTS across nodes as well. A broader discussion among the runtime community is needed to identify multiple mechanisms used for supporting such migratability. Vendor support of virtual memory mechanisms for enabling migratibility, including the ability to reserve unmapped portions virtual space within processes, is necessary and should be standardized by the runtime system community. An alternate approach is to use compiler support. However, that is outside the scope of the runtime community; also the influence of the HPC community over compilers is limited.

3.2.4 Introspection Service

Since the runtime manages events, local scheduling, and task/data reconfiguration for resilience, it will need to maintain an introspection database. The responsibilities of this component is to maintain this database at the level of detail and completeness required by the other components of the RTS (RunTime System). We expect the local scheduler to provide most of the instrumentation input to this component. Additional input may come from language level components and the application programmer. Low-level hardware may also create events that are recorded. The services it provides may included distributed access to a detailed database as well as summary statistics with different levels of details. The information collected can get stale over time, hence this component will include techniques, such as a windowing mechanism or an API for turning instrumentation on/off, to discard old data. Sometimes applications also have complex phase structure; hence there will also be a need for associating data with different phases to customize decisions in this component. In general, introspection services can help enable adaptive optmization at different levels of the RTS.

3.2.5 Resource Management

Adaptive resource management will be an essential function of the exascale runtime systems. Multiple categories of resources will have to managed, so as to relieve applications and programming languages of the increased programming burdens arising out of exascale concerns. We describe three major services: power management, load balancing, and a cross-cutting adaptive controller service.

3.2.6 Power Management Service

The responsibility of this component includes managing power, energy, and temperature. Optimizing these metrics for an entire machine that is executing multiple jobs is outside the purview of this document. However, the runtime associated with one job (the focus of this document) must engage in a bidirectional communication with a full-machine OS. It may receive specifications for power allocation and may inform the full-machine OS about the characteristics of the job it is running.

Locally, on each node, this component interacts with introspection component to monitor core temperature and power consumption. It interfaces with the low-level hardware controllers to set power levels for components such as CPU, memory subsystems, caches, etc. It may also decide to turn some subcomponents on/off at its discretion. It also interfaces with the load balancers to inform them of the changes it is making and/or coordinates with the load balancers. If future machines allow a fine-grained control of power (e.g., low overhead for changing power levels of individual cores), this component can interface with the scheduling service to allow control of power every time a work unit is scheduled.

3.2.7 Load balancing Service

Load balancers decide which migratable entities should be moved to which nodes/processors and when. Load balancers get their input data from information from the programming abstraction, e.g., how different

units of work may share data, and from introspection/instrumentation components at runtime. The data may pertain to the computational load associated with each migratable entity and possibly the strength of communication between pairs of interacting entities. For scalability, this component must assume that the instrumentation database is distributed among processors. Load balancers must take into account that some of the entities recorded in the database may be work and data units that are (marked as) not migratable. Migratable entities may exhibit varying degrees of persistence in their computational loads and communiction patterns, and the load balancers should take advantage of this persistence as a predictive principle, to the extent feasible. The main responsibility of this component is to decide the new location of each migratable unit. It also decides when to do such location changes, whether to do them incrementally or at once, which strategy to use for making these relocation decisions, etc.

Some migratable entities are transient, so there is no history and no need for persistent data after completion. To avoid confusion with the overloaded term "task", we use the term "TMU" (transient migratable unit) to denote these work units. These entities have different names in existing runtime systems (e.g., Charm++ calls them "seeds"); we use the term TMU to avoid being specific to an existing runtime system.

Typical usage scenarios of such parallelism includes state space search and divide and conquer computations. Load balancers that move TMUs between queues on processors, can be an effective approach for such computations. Strategies for such load balancers, which are distinct from those for migratable entities, will also remain an active area of research, with issues such as trade-offs between load balance and communication reduction as well as scalability. Such strategies include, but are not limited to, various flavors of work stealing–based load balancers, including their distributed memory variants. The interface for TMU balancers involve a call to instantiate a fully described work unit, possibly with associated priorities.

Some applications, especially multi-physics and multi-scale simulations of future, may require coexistence of both types of balancers described above. For example, a bunch of small sub-scale simulations may be spawned as TMUs. The balancers must then cooperate to avoid working at cross purposes. Broadly, the persistent work creates a background load within which the TMU balancers must work.

3.2.8 Locality Discovery and management

This is an important functionality. However, it can be subsumed by the functionality of the local scheduler and the global load balancer. The local scheduler can keep track of the recently executed work units and their data footprints to exploit locality among the memory hierarchy of the node. The load balancer will take locality among interacting components into account in deciding which units to place on the same node. It also should take the interconnection topology into account where appropriate to place units on nearby nodes.

Resource management strategies require information about locality of physical resources: the runtime must provide services for discovery of network topology and within-node "topology"—i.e. affinity and degree of connectivity between cores and memories.

3.2.9 Adaptive Controller Service

The components of the runtime can be tuned or adapted based on the observations made by the runtime itself, by analyzing the introspection database. In addition, the application may provide some knobs for the runtime to tune. A meta-controller component might be necessary to carry out global optimizations across these multiple knobs provided by them. This component is also responsible for mediating between conflicting recommendations by different components, e.g., between power manager and load balancer.

3.2.10 Naming Service

As migratable entities (data units, work-units, etc.) are dynamically created, they need to be given a name that is globally valid and unique. Schemes for generating such names fall under the purview of this component. Two types of functionality are supported by naming services: a scheme for creating systematic names (bit-patterns) that does not need global communication, and a global split-phase service for creating a unique name.

Names are global in the sense that they may be accessed by appropriate work entities or as elements of structures anywhere in the physical system allocated to the executing application. The logical elements to which global names are associated may be ephemeral and therefore have a finite creation and termination event associated with them. This implies that the unique names are also ephemeral. Therefore, ultimately a method for name encoding may have to include a method of reuse. Access by some program elements to other globally named objects may be modulated by protection policies and access rights (e.g., capability-based addressing) that will preclude some access or types of actions that can be performed on the named objects. A challenge for the naming service is to permit named abstract objects to be migrated across physical domains (e.g., nodes) while leaving the global name unchanged. This will be facilitated by the location management service (Section 3.2.11). While challenging, such a capability is required for load balancing, reliability reconfiguration, dynamic resource allocation by the OS to the runtime, and dynamic adaptive execution. Global names may include structure to represent hierarchy. An element of a structure may be a first-class object as is the entire structure itself. The name should represent the relationship of the named element within the parent structure. This is essential for affecting vertices in graphs which may be dynamic and elements within matrices among others.

3.2.11 Location Service

The responsibility of this component is to find the location (node) given the name of an object/entity. This should have two query interfaces: a blocking (local) interface and a split phase interface. The local interface is allowed to return "unknown" for a given name, which may lead other components of the runtime to use the more expensive split-phase interface. Possible implementations include distributed hash tables, such as in Charm++ and OCR, with possible replication and/or caching of partial information. In addition to the query interface, there should be an interface to the runtime system to inform this component of changes in location of migratable units; in particular, this component needs to interact closely with the load balancing system, as the latter system is often responsible for moving units. Some options for specification of this component include whether to allow it to provide stale information, or to have a split phase interface for accurate information, or an option for selecting either behavior. The task of actual migration of entities across nodes may be carried out by the location manager itself, or by the load balancer, or by co-operation between the two services.

3.2.12 Communication Service

Communication performance has been the primary consideration in programming model scalability, where software overhead, end-to-end latency, injection bandwidth, and bisection bandwidth can all be important metrics depending on application characteristics. While the transition from terascale to petascale applications was primarily concerned with scalability of the network itself, the transition to exascale will be at least as much about how the increased on-node parallelism interfaces with the network. For example, hybrid programming models involving messaging and threads can suffer from a significant serial bottlenecks if only a single thread is active during communication. Alternatively, if all threads are allowed to communicate, some type of synchronization is needed to control access to shared networking resources, which can also limit performance. Flat message passing models will often scale better than a hybrid model, although they must also manage access to shared hardware resources and suffer from large memory footprints. One-sided communication may have lower overhead than two-sided, because it does not imply synchronization, although many applications require that synchronization and therefore work well with a two-sided model.

The dominant communication layer used in scientific applications is MPI, and that standard continues to evolve based on results from the runtime research community (e.g., one-sided communication) and application requirements (e.g., the need to support multi-physics applications). The exascale challenges described above are of ongoing interest, but exploration of solutions that are not restricted to the current standard are important to ensure that innovations in the systems can be used by applications rather than hidden in a kind of lead common interface scenario. In addition, several programming systems use communication mechanisms that are not based on MPI, e.g., UPC, Co-Aray Fortran, Charm++, and applications like NWChem and Gaussian. These communication layers may better support one-sided communication, remote synchronization and memory allocation, active messages, and asynchronous event-based computations. These may

in turn allow for new types of applications and algorithms that rely on fine-grained asynchronous updates for graph algorithms, global work stealing, or dynamic DAG scheduling.

Although the communication layer has been subject to standardization in the form of MPI, ongoing research into communication runtimes is important to ensure they can perform well on future hardware, namely avoid node level bottlenecks on managing communication as the on-chip parallelism scales, and support new classes of application that push the boundaries of applications to ones that are more dynamic, more fined-grained, and in general less predictable.

3.2.13 Resilience Service

There are a wide range of proposed approaches to address resilience challenges in exascale systems, exploiting knowledge from all levels of the stack, including scientific domain, application, programming system, runtime, hardware, etc. These techniques vary widely in the level of programmer effort and semantic knowledge exploited. Failure modes addressed may include fail-stop failures as well as Silent Data Corruption (SDC). The runtime architecture must support this wide range of resilience techniques. Many strategies depend on the RTS's reliable data store which could be realized by file systems, memory replication, NVRAM, etc. A single standardized interface for all resilience techniques seems infeasible; we enumerate three types below. Yet, it seems desirable to identify all potential contact points between fault tolerance protocols and other runtime components.

Checkpoint restart: Basic widely used resilience service. It depends on the reliable data store. Examples include SCR, FTI, BLCR, etc. It also includes in-memory double checkpoint schemes, which can use local storage such as NVRAMs in addition to memory.

Message logging: Instead of sending every process back to its checkpoint, message-logging scheme sends only the entities housed on the failed node/s back to their checkpoint. Other entities can continue execution subject only to dependences, and if they have to wait, they at least do not consume resources. This speeds up recovery, which can also happen in parallel. A case can be made that for handling fail-stop faults, such schemes will be the main candidate at exascale, assuming high failure frequencies.

Application-oriented, flexible recovery: Libraries that allow application or programming system to manage coverage and overhead of resilience. Typically exploiting applications, model, or other knowledge for efficiency. Examples include GVR (global view resilience).

Selective replication for SDC: Silent data corruption has been identified as a serious problem at exascale by many researchers. Replication schemes may be needed as a "gold standard" but are inefficient by definition. Active research is needed on synthesizing schemes that limit replication costs (both data and computation) while providing protection against SDCs.

Many of the schemes above can use (or may depend on) the notion of a reliable data store.

Reliable data store: For virtually all resilience techniques, an essential element is scalable, high-performance reliable storage of data. File systems are one universal example, but others might include novel systems based on in-array NVRAM, burst buffers, in-memory replication, and so on. The requirements of novel resilience approaches require and will produce reliable stores with latencies much lower (microseconds) and bandwidths much higher (100 PB/s) than even future file systems. Such a store is a critical enabler of x-stack programming systems for resilience.

Chapter 4

Research Case Studies

4.1 DEGAS Runtime Research

As part of the DEGAS project, the DEGAS team has focused on Adaptive Runtime Systems (ARTS), resulting in an adaptive runtime for hierarchical, heterogeneous and asymmetric manycore systems. The ideas used in ARTS focused on integration of tasking and communication to enable hybrid programming, and annotating the runtime with performance history and intentions in order to guide runtime optimizations and enable adaptation. DEGAS has leveraged existing runtimes to achieve these goals, such as the Habanero-C runtime from Rice and BUPC from LBL, as well as the Lithe scheduler composition from UC Berkeley.

As part of the DEGAS project, the team has developed a novel runtime to support HabaneroUPC++, a unified tasking/communication model supporting both one-sided communication (similar to UPC) and intra-node tasking (similar to Habanero-C). This model is completely library-based and does not require compiler support. Instead, it uses the C++ 11 lambda expressions to enable the programmer to write programs that combine UPC communication and Habanero tasking using a syntax that is not overly verbose and similar to what a DSL that combines communication and tasking might look like. The runtime uses the GASNet communication system and a dedicated Habanero communication worker to enable communication among tasks on different nodes.

4.2 D-TEC Runtime Research

While the D-TEC project has mostly been focused on the compiler support for scalable domain-specific languages, relying on other runtimes to manage and execute the compiler-generated code, there were several runtime-related research topics that were addressed as part of the project.

- Runtime support for computation disribution, data distribution, memory management and syncrhonization was added to existing MPI and OpenMP implementations in order to support the execution of programs written using ROSE-supported DDSLs.
- Integration of the X10 and ROSE compilers creates a foundation for targeting X10 runtime by the DSLs supported in ROSE.
- Runtime integration of MPI-3 non-blocking collectives with the X10 one-sided messaging, used to evaluate the scalability and performance of the LULESH proxy application in X10.
- Performance evaluation and tuning of LULESH running on top of the X10 runtime on Cray XE6 at NERSC
- Used ROSE to translate OpenMP code into threaded code that uses the XOMP runtime

4.3 Traleika-Glacier Runtime Research

One of the major research thrusts of the Traleika-Glacier project was the design and development of the Open Community Runtime (OCR).

OCR is a Traleika-Glacier open-source implementation of a set of APIs that define the primitive capabilities that an exascale runtime needs to posses. It is a result of a multi-institutional collaboration between Intel, Rice, UIUC, UCSD, PNNL and Reservoir Labs, with inputs and suggestions from other research groups.

The main ideas reflected in OCR are:

- A program is a collection of dynamic Event-Driven Tasks (EDTs in OCR).
- Program data is a collection of *data blocks* (DBs in OCR). Data blocks are movable and can be distributed.
- Coordination among tasks and data blocks in an application is achieved by explicitly creating *dependencies* among EDTs and DBs using *events*. An EDT executes only when *all* of its input events are satisfied. This is in contrast to other runtimes where barriers or sequential execution may be used to satisfy dependencies, instead of explicit events as in OCR.

OCR runs on a variety of shared-memory and distributed-memory platforms based on Intel x86 and x86_64 processors, Intel Xeon-Phis as well as on Intel's Traleika-Glacier exascale architecture simulator that was designed as part of the Traleika-Glacier project.

While OCR is not designed to be used as a programming model and to write applications directly, there already exist several well-known HPC applications written in OCR that demonstrate its flexibility and effectiveness. Many higher-level approaches to parallel programming, such as Concurrent Collections, Hierarchically Tiled Arrays and the RStream compiler target OCR. There is an ongoing work to allow Legion to run to top of OCR, while efforts are underway to provide a simple parallel language abstraction, called Auto OCR, to target the OCR APIs. Intels FastForward2 project has also demonstrated legacy C and C++ applications, as well as a subset of legacy MPI, to run on top of OCR. Today OCR has an active community developing runtimes for exascale machines in multiple projects across the world.

Selected publications related to OCR are [10, 11, 8, 14, 46, 5, 45, 13, 40, 20, 21, 41, 44, 25].

4.4 XPRESS Runtime Research

XPRESS project has used the ParalleX execution model from Indiana University and the HPX runtime system software from Indiana University and Louisiana State University as the key components of their software stack. Following are several runtime-related research thrusts that are completed as part of the XPRESS project:

- Built the interface between the operating system (LXK) and the runtime that exposes the critical resources to the HPX runtime sytem.
- Dynamic resource management and task scheduling within HPX
- Runtime Interface to OS (RIOS) definition and description of the interaction between HPX and LXK.
- Interfacing existing legacy applications written in MPI and OpenMP with HPX runtime by using HPX futures and data-flow support, as well as the OMPTX runtime (based on HPX) as a HPX-aware OpenMP replacement.
- Autonomic Performance Environment for eXascale (APEX) is a feedback and control library for performance measurement and runtime adaptation, and allows the runtime to monitor the program execution and implement adaptation policies (such as low-power or high-performance). Using APEX with low-power policy results in significant energy savings on LULESH with minimal performance degradation.

4.5 PIPER Runtime Research

PIPER project focuses on performance analysis for the exascale systems, and as such does not directly involve runtime research. However, performance analysis is a crosscutting activity and interfaces with the runtime as well as with the other components of the software stack. In particular, the dynamic adaptation and tuning tools have to co-exist and interact with the runtime to enable runtime analysis and adaptation. The performance/energy analysis tools have to understand the runtime decisions (such as load balancing and dynamic scheduling) and their impact on performance/energy. On the other hand, the runtime tuning mechanisms have to understand the results of the analysis in order to make correct decision when executing an application.

4.6 SLEEC Runtime Research

While SLEEC project focuses primarily on static, compiler-based analysis and optimization of semantic-rich libraries, there are some aspects of the project that interact with the runtime.

In their hybrid static/dynamic approach for executing programs on GPUs, they augment libraries with information about what data needs to be read/written, and any data transformations that need to be performed. This information is used by the runtime to track data and eliminate unecessary data movement, essentially treating the GPU memory as cashe for GPU computations. Data is tracked at the library granularity and automatically transformed (row-major vs. column-major) for the device (CPU vs. GPU). This approach keeps data up-to-date on both CPU and GPU and avoids additional communication, resulting in significant performance improvement. They also have an extension to this approach that allows the runtime to manage multiple GPU memories at the same time. This presents an additional challenges of decomposing and communicating the data amongst different GPUs, task synchronization across GPUs and preserving data representation on multiple GPUs for complex data when that data is decomposed.

4.7 Corvette Runtime Research

The Corvette project has, in one of its subprojects, focused on data-race detection in applications with low over- head. This allowed them not only to detect insidious non-determinism bugs, but also enabled them to remove unnecessary barriers from NWChem, resulting in significant performance improvements. They have also developed a program monitoring and code generation infrastructure with low runtime overhead for UPC, that is complete, precise, reproducible and scalable.

In other work, they have developed automatic dynamic analysis of floating point precision that allowed them to save time, memory and energy when executing floating point applications while still getting an äcceptableänswer. This approach could be applied in an introspective and adaptive runtime that combines the analysis and execution of floating point applications.

4.8 X-TUNE Runtime Research

X-TUNE project focused on compiler research for heterogeneous platforms and did not investigate runtime techniques. Some of their ideas could potentially be applied in a runtime that uses Just In Time compilation to generate device-specific code.

Chapter 5

Transition Opportunities

This chapter identifies runtime components/services that could be transitioned into production use in a staged manner.

5.1 Hardware Abstraction Layer (Node)

The node level hardware abstraction layer will allow programmers to write device-independent, high performance applications by providing the functionality of Operating System calls to hardware at every node while implementing these services at nodes appropriate to that service. Many node services require inter node cooperation and the runtime should manage this interaction dynamically offering the user a nodal view of the system. Finally for hardware reasons the management infrastructure may be regional (heating in a region of a chip) rather than nodal and the runtime needs to be able to manage this additional complexity. In the following sections we will illustrate the function of the runtime hardware abstraction layer by discussing a few key services.

5.1.1 Node configuration

As a program is initiated or progresses on the machine the runtime must be able to query the state of the system, identify available resources, configure them and make them available to the program. At the point of query these resources may be in any state, including powered down. Since a node that is powered down is unable to participate in such a query, we will need active service nodes in the system that are able to respond for regions of the machine that are inactive and are able to activate them. Each type of node has a specific instruction set architecture that represents the primitive operations of that node. The runtime must provision this node with the necessary software, services and tools to participate in executing the program. In a system with millions of nodes this provisioning should be dynamic and sparing giving the node the minimum capabilities necessary to run the program tasks it is allocated.

5.1.2 Node migration

The runtime must be able to migrate computation (tasks) on a node appropriately. This migration may be initiated by faults, to meet a goal or to adjust performance. The first task is to assess the state of the current node, then make the decision to move, followed by configuring the receiving node and finally transferring tasks. Depending on the urgency and amount of state involved, the migration could be achieved by re-starting tasks on the new node and killing them on the old, waiting for tasks to finish and starting new ones on a new node or by some kind of local checkpoint and restart. All this must be achieved seamlessly in the runtime.

5.1.3 Execution

Program execution requires that the runtime matches tasks to nodes. Some nodes will not be able to complete some tasks and must be excluded from the node pool before selection and configuration e.g. they require a device that is not connected to that node. Some nodes may be capable but not ideal e.g. task would benefit from an accelerator that is only available on some nodes. Some nodes may be ideal for the task but in an inappropriate state of located poorly for the program e.g. the program is running on a rack in one machine and the nodes in consideration are in another rack. The runtime will need to be able to make scheduling decisions that do a reasonable job of using resources efficiently.

5.1.4 Memory Management

Memory allocation will require that the runtime understand memory properties, locality and connectivity. Memory in an exascale system will come in a varying range of performance from registers to spinning disk. Varying functionality including coherent, non-coherent and scratchpad. Some will be volatile and some non-volatile. The runtime will need to provide memory management and garbage collection that understands the properties of the memory available and uses them appropriately.

5.1.5 I/O

The node runtime should make access to I/O services transparent to the user, this can be done by running tasks on nodes that provide the relevant service or by providing a service stub that passes a message to a node that has the service. Some I/O services such as visualization may offer parallel aggregation of messages from nodes to construct frames and parts of frames in parallel assembling them before display.

5.2 Hardware Abstraction Layer (System)

The runtime takes best advantage of the existing underlying computing hardware including processing, memory hierarchy, and networking among other capabilities. The operating system supports the underlying hardware and presents an abstract machine to the runtime that is somewhat less specific to the architecture peculiarities of a given system. It is anticipated that some evolutionary changes to the hardware architecture and therefore the abstraction layer will be driven by the needs for exascale, and therefore some assumptions are made here that at best can be satisfied with shim software, although not to the necessary standards of overhead and latency (and therefore, indirectly, granularity and parallelism).

The runtime system expects a hardware abstraction of executing threads resources that support key capabilities. While hardware/OS threads themselves may be heavyweight, they must support the runtime notion of lightweight user threads that have the following properties:

- Very efficient user thread creation and termination.
- Rapid context switching of user thread including hardware support for preemption (not OS interrupt).
- Thread name space address management support including user threads as first class objects.
- External event synchronization within thread context.
- Intra-thread dataflow control and internal synchronization.
- Timing of thread progress-made-good and time-to-completion.
- Thread error detection from OS, Runtime, and Hardware.
- Thread state duplication in memory off node for resilience.
- Thread clock rate control for side-path energy suppression.
- Hardware may add physical thread resources upon runtime request or require runtime to relinquish use of resources upon demand.

- Runtime is free to apply hardware resources provided through its own internal scheduling policies.

The runtime system operates in a hierarchical global name space abstraction supported by a global address space that has hardware/OS support. The hardware abstraction and interface to the runtime system has the following attributes:

- Satisfies the need of a global name space with migrating first class objects
- Provides a virtual address space both within and across nodes of highly scalable multi-core multi-node systems
- Maintains to the hardware/OS control privileges for address management, allocation, and garbage collection
- Support for and maintenance of a hierarchy of protected contexts with parent-child tree structure each of which embodies a part of the application virtual address space and determines scoping of relatively global variables.
- Any such context will be able to span multiple system hardware nodes. Such abstract contexts are protected by hardware through the equivalent of capabilities based addressing for protection and fault detection.
- Hardware facilitates protected access to state of other contexts (not parent-child sequence) via methods.

The interface between the hardware/OS and runtime instantiations will work both ways, with the hardware able to pass information to or make service requests of the runtime system software hierarchy by local hardware components (e.g., nodes), sometimes to promote their effect from local to global. A runtime context can be invoked by the hardware locally that spans a few, many, or all nodes of the system. The hardware/OS can give such runtime contexts special protected privileges associated with the kernel. Through these same means, the hardware can convey critical information to the runtime concerning resource availability, fault detection, hardware status like processor core clock rates, memory address space blocks, and export network access protocol images. A partial list of the kinds of information flowing from the hardware to the runtime system through the hardware interface abstraction could include:

- Message packets arriving over the hardware network(s) from other nodes of the system.
- Packets arriving from external I/O devices for the application runtime.
- Management of secondary storage interface and exporting file system or equivalent to the runtime system.
- Hardware operational status including processor core rates, voltage, utilization, configuration, fault anomaly history, failed components, etc.
- Requests for runtime system services and creation of runtime contexts for hardware/OS support.
- Interoperability ports with other unrelated executing programs including those of different execution models outside the runtime system hierarchy control or name space.

5.3 Operating System Interface

At the node level, the operating system exposes resources for the runtime system to manage and provides fundamental services, such as protection and isolation. Hardware resources can be virtualized, so the operating system must coordinate with the runtime system to manage the virtual-to-physical translations, especially where the hardware only provides privileged access to these mappings. For some hardware resources, it is appropriate for the operating system to dedicate them solely to the runtime to manage, with an expectation that the runtime system will manage the hardware directly. Runtimes interface to the operating system through a variety of direct and indirect mechanisms. System calls, or traps, are a direct interface

for user-level requests of the operating system. There are also implicit interfaces between the runtime and the operating system that are event driven, such as handling interrupts or faults through abstractions like signals and signal handlers. These interfaces and mechanisms are an active area of exploration for adaptive runtime systems. Issues like the blocking nature of system calls and the cost of using event-driven mechanisms are already known to create unacceptable overheads and provide insufficient support for exploiting the capability of adaptive runtime systems. New interfaces and abstractions are needed for lowering the overheads associated with system services and for exploiting new hardware mechanisms such as user-level interrupts and hardware-level support for suspending and resuming user-level threads.

5.4 Programming Language Interface

The runtime system provides abstractions that implementors of individual programming languages will use. The runtime services described in section 4 can be used by the implementations of multiple programming models. A common runtime achieves two separate objectives: it avoids duplication, so that implementors of each programming model don't need to implement similar services separately. Secondly, it allows for flexible interoperation of multiple programming models. The latter is especially important for programing models that generate data-driven (or message-driven, or macro-data-flow) execution patterns. In a multi-module program written using multiple languages, it allows entities from multiple modules to interleave their execution on a processor without explicit user intervention.

The interfaces provided to language runtime include (a) creation of new named entities—work-units and data-units and restrictions on placements, if any (b) creation of communication ("messages") directed at named entities, or physical resources such as nodes or cores (c) identification of phase boundaries and synchronization points, to facilitate runtime functions (d) explicit triggering of services such as load balancing or checkpoints (in addition to implicit actions by the runtime itself) (e) registration of control points or knobs along with description of their effects, so that the runtime can reconfigure the application to optimize specific metrics (see section 3.2.9) (f) registration of names and attributes of entities visible to the runtime, for the purpose of tagging trace/performance data.

Chapter 6

Conclusions

This report summarized runtime system challenges for exascale computing, that follow from the fundamental challenges for exascale systems. Some of the key exascale challenges that pertain to runtime systems include parallelism, energy efficiency, memory hierarchies, data movement, heterogeneous processors and memories, resilience, performance variability, dynamic resource allocation, performance portability, and interoperability with legacy code. In addition to summarizing these challenges, the report also outlined different approaches to addressing these significant challenges that have been pursued by research projects in the DOE-sponsored X-Stack and OS/R programs. It also included a chapter on deployment opportunities for vendors and government labs to build on the research results from these projects.

Appendix A

Ontology with Examples from Current Systems

Many terms used in the runtime research literature are overloaded, often with overlapping or even conflicting meanings. A prime example of this is the word "task". It is used to denote, among other meanings, (a) a fully described work unit (e.g. a function call with all its arguments packaged) which can be executed on any processor, or (b) a work unit that depends on other data being created/sent by potentially remote entities that executes on the same node where it was scheduled/created.

For this reason, we will define the terms we will use, trying to come up with phrases (possibly new) to avoid overlap and confusion.

Nodes, cores, processors: A machine consist of a bunch of nodes which consists of multiple cores that are capable of sharing memory. We use the term *processor* to cover both nodes and cores, when we wish to say either node or core, at the discretion of the runtime system.

Work unit: A piece of computation to be performed. A "pure" work-unit is fully described, without needing any data from others (see "TMUs"). The general notion of work-unit encompasses those as well as others, including user-level threads (ULTs), migratable objects, dependent execution blocks), tasks of all types, etc.

Data units: These contain tiles (or coarse grained chunks) of data. Each data unit is independently migratable, and may have a global name.

Global name: an ID that uniquely identifies an entity across the entire ongoing parallel computation.

Migratable entities/units: These are the entities that the RTS is free to move between nodes at its discretion. Communication may be directed at them.

TMUs ("transient migratable units," or pure work units): TMUs are fully described, self-contained, work units. Once created, they do not depend on (and don't need to wait for) any external data. TMUs do not have a global name, so communication (messages or whatever we call them) cannot be directed to them.

Dependent Execution Blocks (DEBs): work units that depend on production/availability of local or remote data and/or a synchronizing signal from local or remote work-units. These are units of scheduling on a processor (node or core depending on the RTS), typically under the control of the RTS.

Sequential Execution Blocks: Typically, individual portions of the DEBs can be designated as SEBs. They can be function bodies, loop nests or other segments of code being executed. These are typically the units of instrumentation.

A.1 HPX

H1 Data-driven Scheduling: HPX provides a number of complementary mechanisms for data-driven scheduling, including parcels (similar to closures), compute complexes (similar to tasks), and local control objects (LCOs – of which dataflow and futures are two important instances). The HPX scheduler handles inter- and intra-locality scheduling in a unified fashion.

Project	HPX	OCR	ARGO	Charm++
Data-driven Scheduling	$H1$	$O1$	$A1$	$C1$
Introspection	$H2$	$O2$	$A2$	$C2$
Naming	$H3$	$O3$	$A3$	$C3$
Location Management	$H4$	$O4$	$A4$	$C4$
Resource Management	$H5$	$O5$	$A5$	
Power Management	$H6$		$A6$	$C5$
Load Balancing	$H7$		$A7$	$C6$
Adaptive Control	$H8$		$A8$	$C7$
Communication	$H9$	$O6$	$A9$	$C8$
Concurrency Control	$H10$	$O7$	$A10$	
Termination	$H11$	$O8$	$A11$	$C9$
Resilience	$H12$	$O9$	$A12$	$C10$

Table A.1: Examples of services supported by existing runtime systems. Notes in table refer to the list below.

H2 Introspection: Resource and performance instrumentation has been incorporated into the HPX framework since its inception, and has recently been augmented to include support for other instrumentation systems. The instrumentation is available to the runtime and to programs using the runtime to enable dynamic responses to the program, its data, or the computing environment while the program is running.

H3 Naming: HPX provides a single active global address space whose data are distributed throughout the physical environment. The mapping of data to locality is dynamic, meaning the data can migrate from locality to another without requiring notification of users of that data. In addition, HPX provides a hierarchical name space of localities that enables internal protection, control, and potentially some efficiency optimization opportunities based on the dynamic organizational concept of the ParalleX Process (and ParalleX Procedure). Any first-class objects in HPX can have an address in the global address space and/or a name in hierarchical namespace.

H4 Location Management: Localities in HPX are abstractions of a physical synchronous domain (essentially equivalent to a single memory space or some domain where ordering of operations is definable, even if only weakly). The hierarchy of localities extends this to include a generalized notion of proximity.

H5 Resource Management: Resource management is one of the most important capabilities of any runtime system. Introspection and adaptive execution in HPX allow resources to be used as needed, with computation *percolating* to appropriate localities, following migrating data as needed.

H6 Power Management: A particular instance of instrumentation, resource management, and adaptive control in HPX is power (and/or energy). Power is included as one of the fundamental properties of Parallex execution model on which HPX is based. With the capabilities of HPX, power consumption can be managed with migration of work as well as e.g. frequency scaling.

H7 Load Balancing: Load balancing can be accomplished in HPX with a variety of mechanisms including migration of a compute complex from one locality to another as well as the ability to send work to data using the *parcel* mechanism.

- H8 Adaptive Control: Using its introspection capabilities as input, HPX includes feedback mechanisms for adaptively controlling its use of runtime resources, including computation, memory, and power. Further, the notion of dataflow-based scheduling using LCOs adapts execution to available data, triggering appropriate computation.
- H9 Communication: HPX integrates its communication scheduling with its task scheduling. The actual communication is carried out via a layer called Photon, which enables portability across transports and currently supports IB, Gemini/Aries, and Ethernet. Photon supports efficient remote direct memory access (RDMA) and one-sided, asynchronous operation.
- H10 Concurrency Control: Concurrency control in HPX works in tandem with scheduling. The principal mechanisms for concurrency in HPX are lightweight tasks and lightweight active messages (although the use by the programmer and the scheduler of these two abstractions is uniform).
- H11 Termination: HPX provides distributed termination detection functionality to reduce dependence on global synchronization and data exchange.
- H12 Resilience: The HPX resilience model is based on detecting, localizing, and containing errors using a compute-validate-commit cycle. The active global address space provides a natural vehicle for committing changes to memory once a set of computation has been validated.

A.2 Open Community Runtime (OCR)

- O1 Data-driven Scheduling: Tasks are scheduled after their data is made "available" to them so that when a task becomes schedulable, it is guaranteed to have the data it needs to operate.This is different from data-flow scheduling where writing data triggers tasks. In OCR you have to explicitly pass data along an event to schedule a task whereas in dataflow tasks long living and data flows through them.
- O2 Introspection: OCR currently offers limited support for both hardware and application introspection, more is planned as follows. OCR's hardware introspection tracks the state of the hardware (faults, overheating, etc.). This is used in two ways. The scheduler uses it to make sure it does not schedule things on faulty hardware and a separate system modulates the hardware (frequency scaling and what not). OCR's application introspection tracks the state of the application and how it is running and uses it to provide hints to the scheduler (complementing the ones optionally given by the user). This includes things like tracking the number of reads/writes and where they are going to, figuring out the type of operation the EDT is using to map it to better hardware next time, etc. This would therefore include both hints at the application level (horizontal) and at the mapping level (vertical).
- O3 Naming: OCR provides GUIDs which uniquely identify tasks, data and events. In the future it will providing 'GUID tagging' which will allow the user to assign a name to GUIDs in certain cases so that the user can reason about the names (as opposed to them just being a black box).
- O4 Location: The OCR runtime manages locality hiding absolute locality from the programmer. OCR does not currently allow the user to specify "run this task on this core" or "place this data in this memory".
- O5 Resource Management: OCR manages hardware resources turning things on and off and scaling their frequency. This is done without burdoning the user and in future we expect these resource management services to be extended to I/O and storage.
- O6 Communication: OCR provides a communication service that is transparrent to the programmer.
- O7 Concurrency: OCR manages concurrancy through event driven tasks.
- O8 Termination: OCR expects the programmer to explicitly terminate tasks by calling into ocrShutdown() or ocrAbort(). OCR also has a notion of a "finish EDT" which notifies something whenever it and all of its created children have completed.

O9 Resilience: OCR is planning to offer a resiliancy service. There will be an API that supports the EDT and DB model in delivering fine grained control of the resiliency service.

A.3 Argo

The node-level Argo runtime, "Argobots" provides primitives to be used by programming model and language facing runtimes to implement their services. Argobots provides these primitives: (a) Execution Streams (ES) that are bound one-to-one to a CPU core or hardware thread and (b) work units such as User level Thread (ULT) and tasklets. The latter can be inserted into queues (task pools).

A1 Data-driven scheduling: A data-driven scheduler can be implemented using Execution Streams, work units, and event or task queues.

A2 Introspection: Argobots supports event logging for the high level runtime to introspect the execution of applications. Argobots also plans to support hardware introspection by getting and giving feedback from/to Beacon and Expos in the Argo system. Beacon and Expos are backplanes for event/control notification and performance monitoring, respectively.

A3 Naming: Argobots assigns Execution Streams and work units unique identifiers, but leaves the programming model runtime to manage naming of objects.

A4 Location Management: Argobots exposes two kinds of locations, Execution Stream (ES) and pool, to which work units belong. ES can be regarded as a locality domain, and thus work units running on the same ES share the same cache memory. A task pool is a set of work units and is associated with a scheduler. Depending on the type of pool, it can be shared between multiple schedulers, which means that a single pool can be scheduled on more than one ES. With these mechanisms, the user is able to control node-level locality.

A5 Resource Management: In order to adapt to the changes in resources, Argobots plans to work closely with the OS. For instance, when the number of available CPUs cores is changed, it would be able to stop an Execution Stream or create a new one, and to migrate work units. Moreover, it is planned for Argobots to give feedback to the OS on how resources are used.

A6 Power Management: Argobots will expose the power management functionality provided by OS or external libraries.

A7 Load Balancing: Argobots offers the mechanisms for a higher level runtime to implement load balancing: work unit migration, shared pools, events, and stackable schedulers with pluggable scheduling policies. With these mechanisms, the language runtime can use a scheduler that manages load balancing between different Execution Streams.

A8 Adaptive Control: Adaptive control of work units can be accomplished with scheduling features of Argobots.

A9 Communication: Argobots currently relies on external communication libraries, e.g., MPI, to communicate between nodes. However, it is planned to integrate a data communication engine that will support message-based thread activation.

A10 Concurrency: Argobots supports two levels of parallelism: Execution Streams (ESs) and work units. ES is a sequential instruction stream that consists of one or more work units, which are lightweight execution units such as User-Level Threads (ULTs) or tasklets. ULT is an independent execution unit in user space and provides standard thread semantics at a very low context-switching cost. Tasklet is an indivisible unit of work with dependency only on its input data. Both ULTs and tasklets execute to completion, but only ULTs can yield control. Moreover, ES can have stackable schedulers, and it is possible to switch from one scheduler to another with a simple API. Thus, it is possible for Argobots to run different DSLs in turn. For synchronization, Argobots provides some primitives, such as mutex, condition variable, and future.

A11 Termination: While work units terminate when they finish their execution or can be terminated by request when they are not running, ESs need to be explicitly terminated. Argobots provides APIs to control the termination of ESs as well as work units and lets the language runtime handle the termination of objects.

A12 Resilience: this a future goal of the execution model supported by Argobots.

A.4 Charm++

C1 Data-drive scheduling: A data-driven scheduler is at the core of Charm++ runtime from its inception, [17, 2]. It supports FIFO/LIFO and different types of priorities. User-level threads (ULTS) are also supported. ULTs in Charm++ can migrate across processors, which is supproted using multiple alternate techniquers including isomalloc [4].

C2 Charm++ currently maintains 3 separate introspection databases: load balancing database (computation and communication data), projections and BigSim (event traces). In addition its meta-controller records data about evolution of a few key performance attributes.

C3 Naming: Charm++ uses a scheme with long names that avoid communication for generating unique names. It is being replaced with short names, with a name allocation scheme that uses a distributed split-phase scheme as a general fallback, along with a fast scheme that works in common cases.

C4 Location Manager: Charm++ provides a distributed location manager, which uses a combination of caching and distributed hash table [22, 2] to efficiently track locations of globally visible objects. This service is used in delievery of messages (method onvocations) to objects in presence of object migration.

Charm++ has a location manager that answers queries regarding work-unit location. Migration is delegated to the load balancer.

C5 Power Management: Many experimental strategies ([28, 35]) are availablethat optimize power and thermal metrics. E.g. Automatic temperature control, supported by load balancing, is a service for reducing cooling energy [36]

C6 Load Balancing: Charm++ has a suite of plug-in load balancing strategies that migrate objects in response to load balancing concerns, including fully distributed and hierarchical strategies suitable for exascale ([50, 30]).

C7 Adaptive Control: Various control points ("knobs") in the Charm++ RTS as well as applications can be tuned by control system services in Charm++[29, 43] [15, 42].

C8 Communication: [1, 47]

C9 Termination: Starting with an efficient original scheme [38], Charm++ now includes support for multiple termination detection in multiple scenarios [23]

C10 Resilience: Charm++ supports automatic restart on fail-stop failures [52, 9], using a double in-local-storage checkpoint scheme in its distribution. It works for all Charm++ programs, as long as the job scheduler doesnt kill the job on node failures. A message-logging scheme is also available [26], in a prototype form which works for most programs. [49, 51, 31, 27]

Bibliography

[1] R. V. Aaron Becker and L. V. Kale. Patterns for Overlapping Communication and Computation. In *Workshop on Parallel Programming Patterns (ParaPLOP 2009)*, June 2009.

[2] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, New York, NY, USA, 2014. ACM.

[3] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. Abstract machine models and proxy architectures for exascale computing. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 25–32, Piscataway, NJ, USA, 2014. IEEE Press.

[4] G. Antoniu, L. Bougé, and R. Namyst. An efficient and transparent thread migration scheme in the pm2 runtime system. In *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 496–510, San Juan, Puerto Rico, April 1999. Held in conjunction with IPPS/SPDP 1999. IEEE TCPP and ACM SIGARCH, Springer-Verlag.

[5] M. Baskaran, B. Meister, T. Henretty, S. Tavarageri, B. Pradelle, A. Konstantinidis, and R. Lethin. Automatic code generation for an asynchronous task-based runtime. RESPA Workshop, co-located with SC '15, 2015.

[6] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale System, 2008.

[7] S. Borkar, N. P. Jouppi, and P. Stenstrom. Microprocessors in the era of terascale integration. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 237–242, San Jose, CA, USA, 2007. EDA Consortium.

[8] Z. Budimlić, V. Cave, S. Chatterjee, R. Cledat, V. Sarkar, B. Seshasayee, R. Surendran, and N. Vrvilo. Characterizing application execution using the open community runtime. RESPA Workshop, co-located with SC '15, 2015.

[9] S. Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[10] K. Chandrasekar, B. Seshasayee, A. Gavrilovska, and K. Schwan. Improving data reuse in co-located applications with progress-driven scheduling. RESPA Workshop, co-located with SC '15, 2015.

[11] K. Chandrasekar, B. Seshasayee, A. Gavrilovska, and K. Schwan. Task characterization-driven scheduling of multiple applications in a task-based runtime. In *Proceedings of the First International Workshop*

on Extreme Scale Programming Models and Middleware, ESPM '15, pages 52–55, New York, NY, USA, 2015. ACM.

[12] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 58:1–58:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[13] J. Dokulil and S. Benkner. Retargeting of the open community runtime to intel xeon phi. In *Proceedings of the International Conference on Computational Science, ICCS 2015, Computational Science at the Gates of Nature, Reykjavík, Iceland, 1-3 June, 2015, 2014*, pages 1453–1462, 2015.

[14] J. Dokulil, M. Sandrieser, and S. Benkner. Ocr-vx an alternative implementation of the open community runtime. RESPA Workshop, co-located with SC '15, 2015.

[15] I. Dooley and L. V. Kale. Control points for adaptive parallel performance tuning. Technical report, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, November 2008.

[16] S. Imam and V. Sarkar. Load balancing prioritized tasks via work-stealing. In J. L. Trff, S. Hunold, and F. Versaci, editors, *Euro-Par 2015: Parallel Processing*, volume 9233 of *Lecture Notes in Computer Science*, pages 222–234. Springer Berlin Heidelberg, 2015.

[17] L. V. Kale and A. Gursoy. Performance benefits of message driven execution. In *Intel Supercomputer User's Group*, St. Louis, October 1993.

[18] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[19] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar. Habaneroupc++: A compiler-free pgas library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 5:1–5:10, New York, NY, USA, 2014. ACM.

[20] J. Landwehr, A. Marquez, J. Suetterlein, J. F. Manzano, and C. Plata. Pnnl's open community runtime P-OCR: An asynchronous event-driven runtime that finally flies. DOE Booth Poster at SC '15, 2015.

[21] J. Landwehr, J. Suetterlein, A. Marquez, J. F. Manzano, and G. Gao. Application characterization at scale: Lessons learned from developing a distributed open community runtime system for high performance computing. In *Proceedings of the ACM International Conference on Computing Frontiers*, 2016.

[22] O. S. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[23] J. Lifflander, P. Miller, and L. Kale. Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*, February 2013.

[24] R. Lucas et al. DOE ASCAC Subcommittee Report on Top Ten Exascale Research Challenges. February 2014.

[25] B. Meister, M. M. Baskaran, B. Pradelle, T. Henretty, and R. Lethin. Efficient compilation to event-driven task programs. *CoRR*, abs/1601.05458, 2016.

[26] E. Meneses. *Scalable Message-Logging Techniques for Effective Fault Tolerance in HPC Applications.* PhD thesis, Dept. of Computer Science, University of Illinois, 2013.

[27] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kale. Using migratable objects to enhance fault tolerance schemes in supercomputers. In *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[28] H. Menon, B. Acun, S. G. De Gonzalo, O. Sarood, and L. Kalé. Thermal aware automated load balancing for hpc applications. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[29] H. Menon, N. Jain, G. Zheng, and L. V. Kalé. Automated load balancing invocation based on application characteristics. In *IEEE Cluster 12*, Beijing, China, September 2012.

[30] H. Menon and L. Kalé. A distributed dynamic load balancer for iterative applications. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 15:1–15:11, New York, NY, USA, 2013. ACM.

[31] X. Ni, E. Meneses, N. Jain, and L. V. Kale. Acr: Automatic checkpoint/restart for soft and hard error protection. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13. IEEE Computer Society, Nov. 2013.

[32] V. Sarkar et al. DOE ASCAC Subcommittee Report on Synergistic Challenges in Data-Intensive Science and Exascale Computing. March 2013.

[33] V. Sarkar et al. Exascale software study: Software challenges in extreme scale systems. `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf`, September 14, 2009.

[34] V. Sarkar, W. Harrod, and A. E. Snavely. Software Challenges in Extreme Scale Systems. January 2010. Special Issue on Advanced Computing: The Roadmap to Exascale.

[35] O. Sarood, A. Langer, L. V. Kale, B. Rountree, and B. de Supinski. Optimizing Power Allocation to CPU and Memory Subsystems in Overprovisioned HPC Systems. In *Proceedings of IEEE Cluster 2013*, Indianapolis, IN, USA, September 2013.

[36] O. Sarood, P. Miller, E. Totoni, and L. V. Kale. 'Cool' Load Balancing for High Performance Computing Data Centers. In *IEEE Transactions on Computer - SI (Energy Efficient Computing)*, September 2012.

[37] S. Seo, A. Amer, P. Balaji, P. Beckman, C. Bordage, G. Bosilca, A. Brooks, A. CastellAs, D. Genet, T. Herault, et al. Argobots: A lightweight low-level threading/tasking framework, 2015.

[38] A. B. Sinha, L. V. Kale, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, 1993.

[39] G. Steele. How to Think about Parallel Programming: Not!, 2011. "https://www.infoq.com/presentations/Thinking-Parallel-Programming".

[40] J. Suetterlein, J. Landwehr, J. F. Manzano, and A. Marquez. Toward a unified hpc and big data runtime. STREAM Workshop, 2015.

[41] J. Suetterlein, J. Landwehr, A. Marquez, J. F. Manzano, and G. Gao. Asynchronous runtimes in action: An introspective framework for a next gen runtime. In *IPDRM 2016: First Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware co-Located with IPDPS 2016*, 2016.

[42] Y. Sun, J. Lifflander, and L. V. Kale. PICS: A Performance-Analysis-Based Introspective Control System to Steer Parallel Applications. In *Proceedings of 4th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2014*, Munich, Germany, June 2014.

[43] Y. Sun, G. Zheng, P. Jetley, and L. V. Kale. An Adaptive Framework for Large-scale State Space Search. In *Proceedings of Workshop on Large-Scale Parallel Processing (LSPP) in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2011*, Anchorage, Alaska, May 2011.

[44] 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.

[45] N. Vrvilo. Cnc for tuning hints on ocr. CnC Workshop, 2015.

[46] N. Vrvilo and R. Cledat. Implementing a high-level tuning language on the open community runtime: Experience report. RESPA Workshop, co-located with SC '15, 2015.

[47] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale. TRAM: Optimizing Fine-grained Communication with Topological Routing and Aggregation of Messages. In *Proceedings of the International Conference on Parallel Processing*, ICPP '14, Minneapolis, MN, September 2014.

[48] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An api for programming with millions of lightweight threads. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[49] G. Zheng, C. Huang, and L. V. Kalé. Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.

[50] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale. Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers. In *Proceedings of the Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, San Diego, California, USA, September 2010.

[51] G. Zheng, X. Ni, and L. V. Kale. A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale. In *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[52] G. Zheng, L. Shi, and L. V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *2004 IEEE Cluster*, pages 93–103, San Diego, CA, September 2004.