Exascale Software Architecture

FASTOS COMMON VISION

Author:
Marc SNIR & Arthur Maccabe & John Kubiatowicz

Version:
V1.3
October 6, 2014
Contents

1 Introduction 1
  1.1 Abstract .............................................. 1
  1.2 Background .......................................... 1
  1.3 Design Principles ................................. 3
  1.4 General Structure ................................. 4
  1.5 Composition ........................................ 5

2 Node-Level OS 6
  2.1 Power ................................................. 9

3 Node-Level Run Time 9
  3.0.1 Execution & Scheduling .............................. 10
  3.0.2 Memory ............................................. 11
  3.0.3 Power .............................................. 12

4 Global OS/R 12
  4.1 Global OS ........................................... 12
  4.2 Enclave Composition ................................. 12
    4.2.1 Gap .............................................. 13

5 Hierarchical Resource Management 13

6 Resilience 15

7 Global Information Bus 16
  7.1 Logical Design ......................................... 17
  7.2 Physical Design ...................................... 19

Glossary 20
1 Introduction

1.1 Abstract

This document outlines a vision for the software architecture of future exascale systems, and describes the contributions of the Argo [1], Hobbes [2] and X-ARCC [3] projects to develop and experiment with key components of this software infrastructure. The document describes work in progress, and is incomplete.

1.2 Background

Systems at exascale performance and beyond will face challenges that will require a new software architecture. Among these challenges are:

Scale: The need to coordinate billions of execution threads and manage the resources they use.

Heterogeneity: The need to manage different types of execution units (e.g., latency-optimized cores, throughput-optimized cores and accelerators); deep memory hierarchies consisting of Static Random Access Memory (SRAM) nearby and remote Dynamic Random Access Memory (DRAM) and Non-Volatile Random Access Memory (NVRAM) that are partly under software control; Non-Uniform Memory Access (NUMA) and possibly non-coherent nodes; etc.

Energy: The need to manage power as a first-class resource, in order to reduce energy consumption.

Resilience: The need to cope with hardware failures that are more frequent and not always detected by hardware.

Variability: The need to cope with continuous change in component performance characteristics, due to power management and fault handling.

Moreover, we anticipate more complex workloads as applications evolve to address the challenges associated with building predictive models of complex systems of systems. These applications will be multiscale and multiphysics, and include ensemble runs for uncertainty quantification. Simulations will also be increasingly coupled with in-situ analysis of simulation data and
assimilation of experimental data. Therefore, we expect that systems will increasingly support Composed application and composed Workflow of applications, with components that may have different hardware and software requirements.

The software architecture of current supercomputers reflects a historical evolution from distributed system software, where, at each step, the least change that could provide necessary functionality was effected. As a result, the current structures are far from optimal. In particular

- Resource management above the node level is machine-global and static: A global resource manager allocates nodes to parallel jobs and configures them for message-passing communication. Neither the allocation nor the internal configuration can be changed.

- Above the node level, the only resource that is allocatable is an entire node: There is no management of power and of network bandwidth and limited management of I/O resources. As a result, parallel applications interfere with each other when using shared resources such as the network and I/O servers; and use dedicated resources (nodes and power) in a wasteful manner, because of the static allocation.

- Errors, such as a system crash at a node or an unrecoverable hardware error, are handled globally, with no flexibility on the handling mechanisms and no support for localized error recovery.

- No good constructs exist for coordinating workflows or composed applications with parallel components.

- Current node OS (Linux or light-weight OS) does not provide the required functionality to manage hundreds of heterogeneous cores and deep and complex memory hierarchies.

- Current node OS is designed for the isolation of independent time-sharing processes and fair resource allocation to those; it is not designed for the management of resources shared by multiple tightly coupled cooperating threads within one protection domain.
1.3 Design Principles

We believe that the requirements for exascale and beyond require a new design for the Operating System & Run-Time (OS/R) software architecture. We outline below some of driving design principles:

Hierarchy: The design should be hierarchical, in order to provide scalability. This includes resource management and fault management.

Runtime, rather than kernel To the extent possible, the management of resources within one tightly coupled, parallel application should be done by a user-space run-time, not kernel-space OS code. Kernel-space code should be used only to provide access to resources shared by independent jobs.

Quality of Service (performance isolation) To the extent possible, resources shared by competing parallel applications should be managed so as to provide quality of service guarantees: The alternative to Quality of Service (QoS) is over-provisioning so as to make conflicts rare. Over-provisioning is too expensive in HPC and current systems suffer from conflicting use of shared resources, such as network and I/O servers. Note that this requires hardware and software support to make these resources allocatable; e.g., capacity-limited virtual channels in the network to allocate bandwidth or QoS provision by I/O servers.

Application control: To the extent possible, resources shared by cooperating agents within a parallel application should be managed by the application itself (or its runtime), taking into account the interdependencies between these agents.

Adaptivity: More resources should be expected to be allocatable at finer grain: CPU time (on different types of CPUs), memory space (on different types of memories), power (at different levels of the hardware), network bandwidth, I/O bandwidth, etc. The management of these resources should be coordinated to ensure that decisions are consistent; it should be hierarchical, for scalability; and it should be driven by a continuous feedback from system introspection, in order to handle changes in the underlying platform and application dynamics. In other words, resource management has to be thought of as a hierarchical control system, with a feedback loop that uses monitoring information from the
system to adjust resource allocation; static information on the characteristics of an application and information from past runs can also be used.

**Fault Isolation:** To the extent possible, error propagation should be constrained, so as to support local recovery. Errors should be handled as locally as possible.

**Customization:** The same platform will support very different workloads, requiring different software environments and, possibly, different hardware environments. The system should be designed so as to support customization of the environment in different parts of the system, including OS customization.

### 1.4 General Structure

The general structure of the envisioned software stack is shown in Figure 1. The resources (cores and memory) at each node in the system are partitioned into containers. A container is dedicated to one application for a period of time. Cores within a container normally communicate via shared
memory. Each node will also normally contain service cores that are dedicated to various service functions. An enclave consists of multiple containers, typically communicating in user space via message-passing or Direct Remote Memory Access (DRMA). An enclave consists of a set of resource allocated to a parallel application by the global OS. Optionally, enclaves may be hierarchical, with an enclave OS that allocates resources to sub-enclaves.

1.5 Application Composition

A composed application may involve multiple communicating enclaves. Composition can be based on a workflow that runs multiple application simultaneously, or multiple application in succession, or a combination of these approaches. Support for application composition introduces several challenges:

- The different application components used in a composition may have been developed for different computing environments. To address this challenge, one needs to support a diversity of operating and runtime systems.

- Composition requires communication between the applications. Workflow systems typically delegate this communication to the file system, resulting in unnecessary serialization or and/or unnecessary communication. Mechanisms aimed at bypassing the file system can improve overall system performance.

- Applications used in a composition must be sufficiently isolated to preserve application semantics; however, because these applications are part of a single composed application, there will be opportunities to carefully manage how the needed isolation is implemented.

- Components of a composed application may have different life-cycles: A workflow may compose persistent services with computations that are dynamically spawned at various stages of the workflow.

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. Such a global OS would map high-level abstractions (e.g., the pipelined execution of a list of parallel components) into physical instantiations that can include
the allocation of time $\times$ space blocks of compute resources (nodes or cores), communication resources (network bandwidth and buffer space), and power.

## 2 Node-Level OS

Figure 2 provides a schematic description for a node in a future exascale system. The salient features are:

- Hundreds of cores; the cores may have different performance characteristics, different Instruction Set Architectures (ISAs) and different functionalities
- Deep memory hierarchy, with multiple levels of caches and/or scratchpads and memories of increasing size, increasing latency, and decreasing bandwidth; some of the memory may be non-volatile.
- Non-uniform access time to caches and memories (Non-Uniform Cache Access (NUCA) and NUMA)
• Possibly non-coherent caches and increased software control of data movement

• Integrated intelligent **Network Interface Controllers (NICs)** that offload part of the protocol for user-space communication

The concept of a ‘**node**’ can be ambiguous, as distinct “logical” nodes may be packaged together, for cost and density reasons. To remove the ambiguity, we define a node as consisting of a set of general-purpose cores that can execute in user or privileged mode and have all coherent access to shared memory. The node may contain additional resources that are controlled by these cores, such as accelerators, non-coherent memory, network interfaces, etc. Such a node is controlled by a **Node OS**, that executes on some or all of the general-purpose cores.

The general structure of the **Node OS** is illustrated in Figure 3. A (small) set of **cores** is dedicated to running local OS services, possibly providing a full Linux environment at the node. The service cores are also running the local representatives of higher-level global services – they are running the “node meta-OS”. Each parallel application component runs in a **container** that controls the resources allocated to that component. One major use of such containers is to provide parallel applications dedicated resources. The container resources are dedicated to that application for long periods of time, and allocation of CPU and memory resources within a container is done by the run-time, in an application-specific manner. Interrupts are vectored to the service cores, deamons are running on the service cores, etc.

This general approach can implemented in a variety of ways:
**Localized OS** The OS kernel is configured to ensure that system activities are concentrated on the service cores. A *container* is defined by suitable policies and mechanisms in the OS Kernel that ensures that resources allocated to a container are not interfered with. This is the approach taken by the Cray Linux Environment (CLE) \[8\] and the Tessellation project \[4\]. (The later project focuses on gang scheduling – i.e., the dynamic allocation of resources to containers.)

**Partitioned OS** The OS kernel is partitioned, with some services being provided directly on the application cores and some services provided on the service cores. The fused-OS project \[7\] is pursuing this approach.

**Virtual OS** A virtual machine monitor is used to provide a virtual machine environment to each container. This is the approach followed by the Palacios project \[6\].

The Localized OS and Partitioned OS approaches both provide full Linux OS functionality at all containers; the difference between containers is in performance, not functionality: *General-purpose containers* will provide a conventional Linux environment, while *compute containers* will be specialized for applications making few system calls. The virtualized OS approach provides support for containers with different OS personalities. A compute container will only support a restricted set of Posix calls; those can be satisfied directly by the virtual machine monitor.

While the list above seems to imply discrete, well-differentiated choices, implementations may reside on a spectrum and reuse ideas from more than one of the listed approaches.

The following set of questions will guide the selection of a node OS design:

1. How important it is to have full Linux functionality at each container?
2. How important it is to support different OS personalities (e.g., different Unix kernels) within the same node?
3. How noise-free is a compute container, with each of the proposed approaches? This includes noise due to background OS activities, and noise due to the interference of other containers.
4. What level of protection is needed between containers?
5. How easy it is to dynamically adjust the set of resources allocated to a container? How important is it?

6. How easy it is to communicate across containers? What mechanisms will be provided to do so?

2.1 Power

We expect that nodes will provide differential power control for various subsystems on the node – e.g., the ability to speed-up, slow-down of shut-off each core separately. Power will be one of the resources allocated to a container, and the container will be able to manage its power budget.

3 Node-Level Run Time

The node-level runtime provides a “hardware-facing” view of the resources within one container. It is expected that different programming model specific runtime libraries would be implemented atop the node-level runtime. While the runtime will be functionally available in any container, it is likely to be most efficient when executing in the compute container.

The general node runtime design is illustrated in Figure 4. A container instance consists of a nested hierarchy of domains. A level 0 domain contains a set of Execution Unit (EUs). Each execution unit executes a sequential stream of instructions and is guaranteed to make progress. We normally think of execution units as physical entities – hardware threads; but, conceptually, they could be virtual entities. A domain contains private memory that is accessible only to EUs in that domain, and local shared memory that is accessible to all execution units in the container, but is accessed more efficiently by EUs in the same domain. More efficient access may mean higher bandwidth, lower latency, lower overhead for fences, etc. In addition, EUs may be able to access external memory using noncoherent load/store operations or put/get operations. Similar access (noncoherent load/store or put/get) could also be available for the access of private memories in other domains.
Figure 4: Container runtime

3.0.1 Execution & Scheduling

EUs execute work units. A work unit is either a light-weight thread or a task. A light-weight thread is similar to a POSIX user thread. Its state is defined by a stack, a program counter and a register set. A light-weight thread can execute blocking calls and be descheduled and rescheduled (its state is saved and restored). However, unlike POSIX threads, light-weight threads are not managed by the kernel scheduler, but by a user-space scheduler. A task is, essentially, a function. It cannot block and always executes to completion. Hence, its state is defined by a function pointer and a set of arguments. Work units can spawn new work units.

A light-weight thread can be in one of three states: running, runnable or blocked; a task can either be running or runnable. A blocked light-weight thread is associated with a glssynch. When the event occurs, the thread becomes runnable. Synchronization events are caused by operations on synchronization objects such as locks, or counters. A runnable work unit starts running when it is scheduled on an EU and stops running when it completes or blocks.

Schedulers are functions provided by the runtime or user-defined. Each EU is associated with one scheduler. Each work unit is associated with a
set of EUs that can execute this work unit. For example, these could be the set of all EUs within one domain. When a work unit completes or blocks, control is passed to the scheduler associated with the EU that was running this work unit. The scheduler picks one of the runnable work units that can run on that EU. In addition, a work unit can directly pass control to another runnable work unit.

Other synchronization objects will be defined as needed.

Observations:

• The above defined mechanisms do not include preemptive scheduling and interrupt handling. The preferred method for handling asynchronous events is to dedicate an EU to the execution of event handlers (i.e., tasks that become runnable when an event occurs). Such event handlers are similar to a first-level (user space) interrupt handler: They must be short in duration and may spawn additional tasks (second level interrupt handlers), if needed.

• Priority-based scheduling can be designed as a policy implemented by a scheduler

• The set of EUs that can run a work unit are assumed to have the same ISA. Additional constraints must be respected to ensure that memory references stay valid if the work unit is migrated.

3.0.2 Memory

System-level paging will not be normally used to allocate memory.

The memory management interfaces include two types of methods:

• Methods for allocating and freeing memory of different types – in memories where allocation is done by software (i.e., not caches).

• Methods for moving data. Data movement methods include

  copying (memcpy variants): Such methods specify a source and destination and, if asynchronous, a mechanism for signaling completion.

  moving: These methods combine copying with deallocation at source.

  caching: These methods include prefetch into a cache, and copying, with allocation at the destination. They also include flush and invalidate.
3.0.3 Power

We expect that nodes will provide differential power control for various subsystems on the node – e.g., the ability to speed-up, slow-down of shut-off each core separately. Power will be one of the resources allocated to a container, and the container will be able to manage its power budget.

4 Global OS/R

4.1 Global OS

The global OS allocates resources to enclaves. This includes nodes, or parts of a node; power; and I/O resources. The allocation can be dynamic, with an enclave requesting or relinquishing resources on its own, or as a result of a request from the global OS. The global OS can communicate with the service OS at each node in order to assign containers to an enclave. Once containers are assigned to an enclave, they are controlled by the enclave OS, and the global OS communicates with the enclave OS representative, which will run on one node of the enclave (possibly, with a backup on another node).

The enclave hierarchy can be recursive, with an enclave OS allocating resources to sub-enclaves. For example, the global OS could allocate resources to an enclave running a workflow, the enclave OS allocates resources to component applications, so as to match the speed at which they produce and consume data.

The global OS will subsume functions currently provided by the system scheduler and resource manager: Enclave management will be more complex than on current systems, since more resources become allocatable: not only nodes, but also power, I/O bandwidth, temporary persistent storage, and, possibly, network bandwidth.

4.2 Enclave Composition

Enclaves can be composed, in order to instantiate a workflow or another composed application. The communication between enclaves will be parallel and either buffered or unbuffered. The communication may be in user-space, but with suitable restrictions to enforce isolation. New APIs will be needed to specify composition modes and communication mechanisms between enclaves.
The connection of enclaves will be mediated by the global OS. The connections can be created when the enclaves are instantiated; or enclaves can connect dynamically, using the global OS as a mediator. The connection mechanisms will provide security and may enforce service levels. Some of the enclaves may have a limited lifetime (e.g., the lifetime of a batch job); other service enclaves may be persistent. The global OS controls to which service enclave a batch enclave can connect; and what level of service it can receive.

4.2.1 Gap

An important driver for this work would be a coordination language that can be used to compose parallel modules. No good candidate seems to be widely used.

5 Hierarchical Resource Management

Resource management in the proposed OS/R is hierarchical: A parent manager allocates dedicated resources to its children; the children manage the resources autonomously, based on constraints and objectives specified by the parent. A global resource manager is in control of the entire system and allocates resources to enclaves. An enclave resource manager is in control of resources allocated to an enclave and allocates resource to sub-enclaves, if the enclave is hierarchical, or to containers. A container resource manager is allocating resources to domains.

Resource management is adaptive: The resource manager continuously receives information on the underlying execution and adjusts resource allocation to improve performance. The resource manager is customizable: It is part of the run-time and its behavior is determined by pluggable libraries. Finally, it is integrated: The different resources are managed in a coordinated manner.

The generic resource manager has the structure illustrated in Figure 5. The control module effects a negative feedback loop to keep the controlled system in a “good” state. It receives periodically information on the state of the subsystem it controls; this may include information from performance sensors, and information from the application itself: For example, the application may report on its rate of progress, if it is periodic; or it can report on idle time spent waiting for a communication to complete. The module peri-
Figure 5: Resource Manager
odically adjusts the state of the system, by changing resource allocation. A translator module processes the inputs from sensors before submitting them to the control module, to clean and normalize them. Similarly, a translator module translates the generic feedback of the control module onto “knob-specific” actions.

The higher-level supervisor module provides introspection on the activities of the control module. It initially sets and periodically adjusts the feedback function used by the control module, based on its estimate of what state is likely to optimize performance. It does so, based on information on the constraints the system must obey (e.g., on peak power and on energy usage); information on the objectives of the computation (e.g., minimum time, or minimum energy consumption); and information on properties of the executing code.

Each distinct resource in the system has its own feedback loop. However, the supervisor module is common: It ensures that the policies for managing the different resources are synergistic.

Power management will be the main target of this control infrastructure: We expect power consumption to be a major constraint on future systems. “Load balancing” could be done more cheaply by reducing the power allocated to underutilized resources and increasing power allocated to overcommitted resources, rather than by migrating data or computations without a change in the speed and power consumption of various units.

APIs need to defined for the interface to the supervisor from the higher levels, for the initialization of the resource manager logic, and for the discovery of the “sensors” and “actuators” that are available in the system.

6 Resilience

It is widely understood that future systems will encounter hardware faults much more frequently. This is due to several factors, including the need to greatly reduce energy consumption, continued increases in the numbers of components in these systems, and increased vulnerability of smaller transistors. Software faults may also be more frequent, because of increased system size and increased complexity.

Addressing resilience presents several challenges:

- We do not have good fault models. While we understand the fundamental sources of faults, we do not have good models for the frequency
of faults or the ways in which these faults will become apparent.

- The tradeoffs for handling or masking faults at different levels (e.g., application, library, runtime, etc) are complicated and it is clear that cost-effective solutions will require flexibility.

- There are no tools for testing or evaluating fault coverage.

The exascale OS/R 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. In addition, the the exascale OS/R will need to be itself resilient.

One key technique to improve resilience is fault-containment and hierarchical error handling. Enclaves will act as containment domains [3]: To the extent possible, errors will be detected before they have propagated outside the enclave and will be recovered by 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 an error at its level.

7 Global Information Bus

The global OS design requires, at multiple places, a Communication and Control (CC) infrastructure that conveys sensing information from the environment and the applications to a control system and conveys commands from this control system to the various system actuators. Such information is needed for performance management and error handling.

Several factors complicate the design of this infrastructure:

Scale: CC needs to scale to the size of an exascale machine

Resilience: CC should (almost) never fail, and should provide high QoS levels

Topology mismatch: The logical structure of the system (e.g., which nodes are in an enclave) may not match the physical structure of the system (e.g., which nodes are in a rack).
In-band, out-of-band integration: Part of the information that is relevant to control functions circulates in-band, through the main interconnect; part circulates out-of band, generated by various control processors and moved through the monitoring network to a system console. This segregation is important to vendors, to ensure the reliability and integrity of the system. However, advanced error-handling will require localized decisions by in-band agents.

Dynamic entity matching: Different layers of the system use different naming schemes: e.g., physical node name, vs. process rank; the correspondence between physical names and logical names may change dynamically; this translation may have more than one level.

7.1 Logical Design

The global information bus provides a reliable and scalable publish-subscribe mechanism: Information on the bus consists of typed messages; the different agents can publish information of certain types and can subscribe to information of certain types. Messages carry additional metadata, indicating their urgency and their expiration date.

The logical structure of the global information bus is shown in Figure 6. The pub-sub infrastructure supports different types of clients:

Sensors that send information either when queried, periodically, or when certain events happen. This includes hardware sensors (e.g., temperature), software sensors (e.g., heartbeats), application sensors (e.g., application-specific progress messages).

Actuators that accept commands. This may include controllable sensors.

Translators that map from one ontology (e.g., physical node number) to another (e.g., logical node number within enclave).

Aggregators that aggregate multiple messages into one

Splitters that duplicate messages

Loggers that store message information.
Figure 6: Global Information Bus
7.2 Physical Design

The physical design of the pub-sub infrastructure will be hierarchical, reflecting the prevalent communication patterns. We expect it to consist of multiple trees, with “pass-thru” connections across the trees; the topology will change dynamically, as enclaves are created and destroyed. Splitters and aggregators will be configured to match the tree topologies.

References


Glossary

 accelerator A core that has restricted functionality but enhanced performance. Because of their limited functionality, accelerators typically need to be (partly) controlled by a general core. For example, an accelerator may not have multiple protection rings, and may not be able to execute kernel code. Accelerators frequently operate in a limited address space..

 address space The mapping of a set of names (addresses) to memory units. The mapping does not need to be one-to-one, i.e., multiple names can map to the same memory unit..

 application Software developed to implement a specific (possibly very complex), standalone functionality. Examples include simulation code that implements a weather or climate model and code that implements a statistical analysis of a data set. (See also composed application, job, and workflow).

 cache A small, fast memory that is accessible by one or few cores. A conventional cache supports in hardware (i) address translation (mapping from physical address to cache address), (ii) memory management (allocating space for entries and evicting entries), and (iii) consistency (ensuring that the cache has the most recent value of a variable). (See also scratchpad).
**composed application** An application that is constructed by coupling multiple applications or modules. Every **job** is a composed application; however, composed applications can include modules that were not developed as standalone applications. An example is a climate application which is composed of modules that implement ocean, land, and atmosphere models. An application may also use persistent service providers (See also application, job, and workflow). 2, 5, 12, 20, 22, 23

**container** A mechanism for isolating resources within a node. More specifically, a Linux container is a mechanism provided by Linux to isolate resources within a Linux image, and provide to each container a separate name space. Unlike a **virtual machine** each Linux container presents the same Linux OS interface. (See also virtual machine). 4, 7–9, 12

**container resource manager** The resource manager that manages the resource of a container. 13

**core** A hardware EU that can execute one or more simultaneous **hardware threads**, with some computational resources (such as ALUs or rewrite registers) shared among these threads. (See also hardware thread). 4, 6, 7

**domain** A set of resources, within a node, such as cores and memory, that share some “locality” properties: E.g., synchronization between cores in the same domain could be faster than between cores in distinct domains; access from a core to memory in the local domain could be faster than access to memory outside the domain. We envisage domains as a nested hierarchy. 9, 10

**DRAM** Dynamic Random Access Memory. 1

**DRMA** Direct Remote Memory Access. 5

**enclave** A set of nodes that are managed collectively. This includes the possibility of hierarchy, with an enclave decomposed into sub-enclaves. An enclave may execute an application, a component of a composed application, or provide a service. 5, 12, 13, 21

**enclave OS** The software that manages an enclave. The enclave OS runs on one or few control nodes in the enclave, and has agents on each node of the enclave. 5, 12, 21
enclave resource manager The resource manager that manages the resources of an enclave.

EU A resource that can execute work units. This will normally be a physical resources, namely a hardware thread. However, if the work units are only tasks and light-weight threads, then a thread can function as a “virtual” EU. The OS has to ensure that an EU makes progress.

global OS The software that manages the entire system, allocates resources to enclaves and handles errors.

global resource manager The resource manager that manages the resources of the entire system.

ISA Instruction Set Architecture.

job A composition of applications assembled to accomplish a task. Applications and parts of applications may be invoked dynamically. Running a ensemble of climate applications is an example of a job. The activities of a job (invocation and coupling of applications) are described by a glsjob control language, e.g., a Unix shell script. The global OS provides the mechanisms needed to implement the activities specified by a job. (See also application, composed application, and workflow).

latency-optimized core A core optimized for the fast execution of one or few threads, at the possible expense of throughput. (See also throughput-optimized core).

light-weight OS An OS that provides the minimal functionality needed to support computation while maintaining the degree of isolation mandated by the policies provided by the broader context (e.g., a runtime, enclave OS, or global OS). A lightweight OS provides minimal abstractions of the underlying hardware and may use a variety of techniques to support applications that need additional services.

localized OS A node OS that supports containers by ensuring that OS code runs on service cores while application code runs on Linux containers, or equivalent structures.
NIC  Network Interface Controller.  

**node** A set of (physical or virtual) resources (e.g., cores, memory, network interfaces) that are managed as an atomic unit. Traditionally, nodes represented the basic unit of allocation for compute and memory in an HPC system, while disk storage is a shared resource. Future systems may have more allocatable resources (power, shared NVRAM storage, network bandwidth, etc.).  

**node OS** The node responsible for managing one node, allocating resources to containers and handling errors.  

**NUCA** Non-Uniform Cache Access.  

**NUMA** Non-Uniform Memory Access.  

**NVRAM** Non-Volatile Random Access Memory.  

**Operating System & Run-Time (OS/R)** The software layers that manage resources at a particular level of the system; provide services to the executing threads they control; handle exceptions occurring at that level; communicate with the higher and lower level OS/R; and manage communication with other instances at the same level.  

**partitioned OS** A node OS that splits OS activities into activities supported at the local (compute) core, and activities offloaded to a remote service core, thereby ensuring low system noise on compute cores.  

**QoS** Quality of Service.  

**scheduler** A program that assigns work units for execution on EU. Tasks and lightweight threads are normally scheduled by a user-level scheduler that has no kernel privileges. Such a scheduler cannot preempt a work unit and, hence, is invoked only when an EU has become idle because the work unit it executed terminated or yielded control. Threads or containers are scheduled by a kernel-level scheduler that can preempt their execution.
**scratchpad** A small, fast memory that is accessible by one or few core, and is directly addressable. Hardware may also provide hybrid cache/scratchpad structure that provide some, but not all the features of a cache. 6, 20

**service core** A core dedicated to running OS and runtime services. By segregating service cores from compute cores one takes advantage of large core counts in order to avoid perturbations in the computation due to “system noise”, i.e., background activities and the handling of asynchronous events. 5

**service enclave** An enclave that provides a persistent service, such a file system. 13

**service OS** The OS services that run on service cores – and are offloaded from compute cores. 12

**SRAM** Static Random Access Memory. 1

**synchronization event** An event that may change the status of a work unit from blocked to runnable. 10

**synchronization object** An object that is used to refer to synchronization events. 10

**task** A work unit that executes to completion without blocking or being descheduled: it interacts with a scheduler only once, when it is initially scheduled for execution. A task is normally defined by a function reference and a set of parameter values. (See also thread, light-weight thread and scheduler). 10

**thread** A work unit that can be scheduled for execution on a core. The OS is aware of the thread and can preempt it – threads are managed by the OS scheduler. (See also light-weight thread and scheduler). 21

**throughput-optimized core** A core optimized for the execution of a large number of operations per cycle, at the possible expense of the execution speed of each individual thread. (See also latency-optimized core). 10
virtual OS  A node OS that uses virtual machine constructs in order to support containers. 8

work unit  A sequence of programmed instructions that can be managed by a scheduler and allocated for execution on an EU. 10, 11

workflow  The description of a set of actions, including their interrelationships, that need to be completed to accomplish a task. Every job is an example of a workflow; workflows may additionally include actions that require resources that are not managed by the Global OS (e.g., the Spallation Neutron Source). (See also application, composed application, and job). 2, 12, 20, 22