Application-Driven Requirements for Node Resource Management in Next-Generation Systems

Edgar A. León[*], Balazs Gerofi[†], Julien Jaeger[‡][**], Guillaume Mercier[§], Rolf Riesen[¶], Masamichi Takagi[†], and Brice Goglin[||]
[*]*Lawrence Livermore National Laboratory, USA* [†]*RIKEN, Japan*
[‡]*LIHPC Université Paris-Saclay, France* [**]*CEA, DAM, DIF, F-91297 Arpajon, France*
[§]*Bordeaux INP, France* [¶]*Intel Corporation, USA* [||]*Inria, France*

***Abstract*—Emerging workloads on supercomputing platforms are pushing the limits of traditional high-performance computing software environments. Multi-physics, coupled simulations, big data processing and machine learning frameworks, and multi-component workloads pose serious challenges to system and application developers. At the heart of the problem is the lack of cross-stack coordination to enable flexible resource management among multiple runtime components.**

In this work, we analyze seven real-world applications that represent emerging workloads and illustrate the scope and magnitude of the problem. We then extract several themes from these applications that highlight next-generation requirements for node resource managers. Finally, using these requirements, we propose a general, cross-stack coordination framework and outline its components and functionality.

***Index Terms*—Scientific computing, Supercomputers, Processor scheduling, Cluster computing, High performance computing, Software performance, Software reusability, System software, Operating systems, Utility programs, Programming environments, Runtime, Runtime environment, Software libraries.**

I. Introduction

In recent years workload diversity in HPC environments has exploded. Big data processing, in-situ analytics, artificial intelligence and machine learning workloads, as well as multi-component workflows are becoming common-place on supercomputers. Each of these workloads bring specific runtime requirements that are pushing the traditional supercomputing software environments to their limits.

In addition, with the end of Dennard scaling and the slowing down of Moore's law, the prevalence of heterogeneous, special-purpose compute devices, i.e., accelerator-based computing, is growing and overall hardware complexity is increasing. The difficulty to efficiently utilize these platforms nurtures interest in multi-tenancy and more dynamic, cloud-like execution environments for HPC centers. These trends force the combined use of software runtime components inside compute nodes that were not designed to cooperate with each other, and often lead to suboptimal performance. The lack of a coordination framework between different runtime components that enables dynamic assignment of hardware resources is at the core of the problem.

To study this problem, we analyze a representative set of emerging applications that test the limits of node resource management capabilities of traditional supercomputing environments. We focus on flexible resource management among runtime components within compute nodes, as opposed to resource management across compute nodes, which we consider part of the global resource manager. Our application use cases highlight the underlying forces shaping the requirements for next generation systems.

We find that hardware resource subsets, e.g., CPU cores, need to be clearly categorized, so that runtime components with different characteristics can be mapped to the appropriate hardware, thus avoiding competition and interference. For example, assigning auxiliary utility threads serving asynchronous background activities, e.g., MPI progress threads, to exclusive parts of a many-core chip can yield significant performance improvements. Current software interfaces make such configuration hard to achieve. With the introduction of nonblocking collective communication in MPI, such functionality could benefit an increasing number of applications [1].

We also find that static resource affinity policies are no longer adequate. Static assignment of compute resources in a domain-decomposed HPC simulation typically works well, but multi-physics, coupled applications that bring together different runtime packages and libraries have changing requirements as the application executes over time [2]. For example, there is a need to dynamically reconfigure the resources assigned to MPI processes and OpenMP threads, but that requires custom modifications to existing runtime components due to the lack of appropriate interfaces [3].

Furthermore, emerging machine-learning workloads that use frameworks such as TensorFlow [4] and LBANN [5], require multiple software components, each of them launching different types and number of workers including CUDA threads, C++ threads, OpenMP threads, and POSIX threads. While there is a need to manage thread heterogeneity, concurrency, and their placement onto the compute resources, current software interfaces lack such functionality.

This work makes the following contributions:

- We analyze seven, real-world, scientific applications that stress the resource management capabilities of current HPC software environments.
- We identify the latent resource conflict themes in these applications and organize them into a taxonomy that enables a better understanding of how they should be addressed.
- We propose a framework for cross-stack coordination

978-1-6654-2268-0/20/$31.00 ©2020 IEEE
DOI 10.1109/ROSS51935.2020.00006

within compute nodes for the dynamic management of hardware resources among runtime components.

Lastly, we include an Artifact Description Appendix describing the hardware and software environments used for the experiments presented in this paper as well as the configurations of the scientific applications.

II. Challenges of Real-World Applications

In this section we describe seven applications that exemplify the problem we are trying to solve. These emerging applications make use of multiple programming abstractions, compute engines, and new software environments that did not evolve on supercomputers. Combined with traditional environments such as MPI and OpenMP, the resulting amalgam is bound to generate resource usage conflicts and sub-optimal performance.

We chose the applications in this section because each demonstrates a particular use case and the unique circumstances that are problematic in current supercomputing environments. After describing each application we call out the particular challenges we identified and summarize the main issue for each use-case. In Section III we discuss these issues in more detail and derive next-generation resource management requirements.

A. Physics and Chemistry Science

GeoFEM [1] and NWChem [6] are applications that have the potential to overlap communication and computation. GeoFEM is a parallel simulation code based on the finite-volume method developed at the University of Tokyo. It simulates ground-water flow problems through saturated heterogeneous porous media and uses a conjugate gradient solver with multigrid preconditioner for solving Poisson's equations. NWChem is an ab initio computational chemistry software package developed and maintained by the Environmental Molecular Sciences Laboratory at the Pacific Northwest National Laboratory. NWChem includes the coupled cluster theory (CC) for accurate quantum-mechanical description of ground and excited states of chemical systems.

Both of these applications have been modified to utilize nonblocking MPI collective operations for overlapping computation and communication. However, the placement of asynchronous communication progress threads of the MPI library plays a critical role in attaining performance.

NWChem running on 32 dual-socket Intel Xeon E5-2680v2 compute nodes, each with 10 cores per socket (hyper-threading turned off), demonstrates the impact of careful thread placement. The application uses 16 cores across 16 MPI processes, spawning one MPI utility thread per process for asynchronous communication progress, i.e., 16 utility threads per node. The four spare cores, which can be used for system services, are used when utility threads are explicitly pinned. Figure 1 shows the execution time of three configurations with respect to the placement of MPI progress threads.

Usage of MPI progress threads for nonblocking collectives does not automatically result in performance improvement and

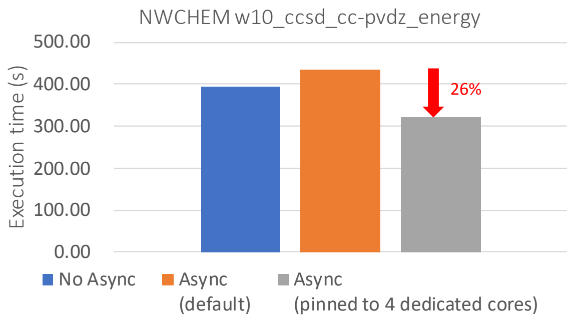

Fig. 1. The impact of utility-thread placement on NWChem.

can decrease performance (blue vs. orange bars). On the other hand, as shown by the grey bar, proper placement of progress threads reduces execution time by 26%. For the sake of brevity we omit detailed results for GeoFEM, but note that we observe similar behavior. Specifying the affinity of these threads is not trivial, because there is no standard interface to control MPI progress thread placement.

Challenges: These application use cases demonstrate the need for a standard API that enables fine-grained control over placement of asynchronous communication progress threads. The challenge is to clearly distinguish progress threads from compute threads and precisely control their placement.

Issue: Dynamic work performed by auxiliary libraries.

B. Climate Modeling

The field of climate science includes the Ocean Physics, Atmospheric Physics, and Marine Biogeochemistry domains. Some domains share similar models but, in most cases, domain-specific tools have been developed. The various models interact by exchanging data with one another. For example, an atmospheric boundary layer modifies surface temperature from the ocean and vice-versa. Each model is integrated into a coupled application through a model-specific kernel. Designed independently, these kernels can be sequential or parallel, use only MPI, Threads, or MPI+Threads, as shown in Figure 2.

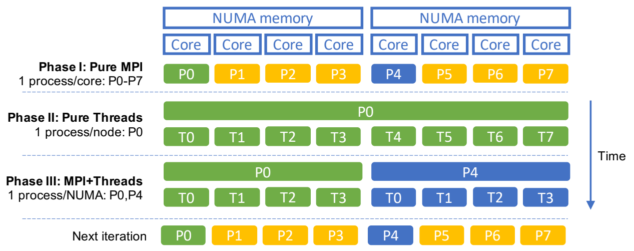

Fig. 2. Kernels or phases of a coupled simulation.

An example of a multi-kernel simulation using the OASIS code coupling software [7] is comprised of two parts: the SCRIP weight interpolation [2], which uses shared-memory parallelism, and the NEMO-BENCH oceanic circulation computation [8], which uses distributed-memory parallelism. These parts can be mapped to Phase II and Phase I (next iteration) of Figure 2, respectively. During the SCRIP phase, the MPI lead process spawns a predefined number of OpenMP threads, while the other MPI processes are quiescent. The number of threads that the SCRIP library launches as well as the number of active MPI processes do not match the specific MPI+OpenMP decomposition later phases need—NEMO-BENCH uses MPI-only parallelism.

Figure 3 shows the simulation's execution time on 2 compute nodes with 40 SMT-2 cores. It includes three configurations varying the number of MPI processes for NEMO-BENCH (NEMO workers) and the number of OpenMP threads for SCRIP (SCRIP workers). This figure shows that the best process-thread heterogeneous configuration is kernel specific: 40 MPI processes per node for NEMO-BENCH and a single process per node with 80 OpenMP threads for SCRIP. This configuration is up to 23% better than the other configurations.

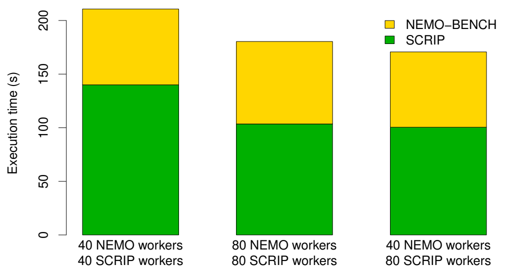

Fig. 3. Cumulative time of a multi-model simulation with dynamic hybrid parallelism. NEMO-BENCH uses MPI-only parallelism, while SCRIP uses a single process per node with multiple OpenMP threads.

Unfortunately, there is no standard interface or programming support to dynamically reconfigure different phases and their hardware mappings, which signifies this process must be performed repeatedly for each multi-kernel simulation.

Challenges: Thread heterogeneity stemming from different models in one application is difficult to program. Because the optimal process/thread configuration is different in each specific model, a single, static configuration can lead to significant overall performance loss.

Issue: Rebalancing and remapping of MPI processes and threads in different application phases.

C. Hydrodynamics

The French Alternative Energies and Atomic Energy Commission employs Arbitrary Lagrangian-Eulerian Hydrodynamics simulations. These multi-phase codes include multiple objects and multiple materials, where each object in the simulation has its own mesh.

The Hydrodynamics simulations expose a similar behavior as SCRIP in terms of resource rebalancing. The global simulation is divided into multiple recurring phases. Each phase, performed in two steps, deals with the contact between the different materials. The first step computes the contact forces between the materials and uses domain decomposition with one MPI process per core. Depending on the materials, some friction displacement needs to be computed in a second step. This step performs best when assigning all the cells to one MPI process and using OpenMP threads. This is due to the dependencies between the cells required for this calculation. However, there is no standard way to switch between these two configurations.

To overcome this limitation, the applications were changed as follows. For each friction phase (Phase II in Figure 2), a lead MPI process is selected. It gathers information from all the cells involved in this calculation, and steals other MPI compute resources which are put to sleep. The lead MPI process launches as many OpenMP threads as hardware threads available. Once the friction phase completes, the mapping of MPI processes on compute resources is restored to one MPI process per core (Phase I in Figure 2).

We ran the application using only MPI throughout the application and compared it with the hand-tuned version that uses only OpenMP on the friction phases. The former ran in 60,771 secs while the latter ran in 32,112 secs. The dynamic reconfiguration of phases provides a $1.89X$ speedup.

Challenges: This use case describes different computation schemes that can appear within one multi-phase application. Like with SCRIP, the optimal process/thread configuration is different in each specific step, and a single, static configuration can lead to significant overall performance loss.

Issue: Rebalancing and remapping of MPI processes and threads in different application phases.

D. Machine Learning in Inertial Confinement Fusion

The National Ignition Facility at Lawrence Livermore National Laboratory is using deep neural networks to steer simulations in Inertial Confinement Fusion (ICF). Novel tournament methods are used to train a single model on vast quantities of data generated by ICF simulations. Components of this model drive speculative sampling to carefully choose which simulations to execute.

These ICF simulations use the Livermore Big Artificial Neural Network (LBANN) toolkit [5]. LBANN accelerates the training of massive neural networks on HPC systems. The toolkit is designed as an MPI+Threads framework. On heterogeneous CPU+GPU architectures, one MPI task per GPU is used. A number of C++ threads on the CPU are used for processing the input and transformation layers; convolution, ReLU, and batch normalization are performed on the GPUs; and the soft max and metric layers are performed with OpenMP threads on the CPU. For communication, LBANN employs the Aluminum library [9] to handle latency-sensitive operations and the NVIDIA NCCL library for bandwidth-sensitive operations.

Figure 4 shows the processes and the various software threads that are launched to perform the compute, I/O, and communication tasks on a compute node with two sockets and four GPUs. There are MPI processes (P0-P3), GPU kernels (K0-K3) and four types of threads: I/O threads (IO) implemented as C++ threads, compute threads (Comp) implemented as OpenMP threads, Aluminum threads (Al) implemented as POSIX threads, and NCCL threads implemented as POSIX threads. Coordinating affinity and binding of all of these threads and processes to minimize interference and improve locality is a significant challenge. Today, this process is done manually and prone to inefficiencies.

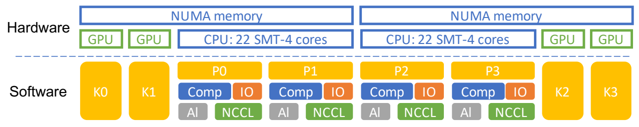

Fig. 4. LBANN on a dual-socket, multi-GPU architecture.

Challenges: LBANN requires multiple components, each with different types and number of workers, including CUDA kernels, C++ threads, OpenMP threads, and POSIX threads. Challenges include managing thread heterogeneity, concurrency, and efficient placement onto the compute resources.
Issue: Multiple, uncoordinated types of threads.

E. Real-time Weather Forecasting

RIKEN developed a high-resolution, real-time weather forecasting system, called SCALE-LETKF [10], to predict severe, short rainstorms in Japan (Figure 5).

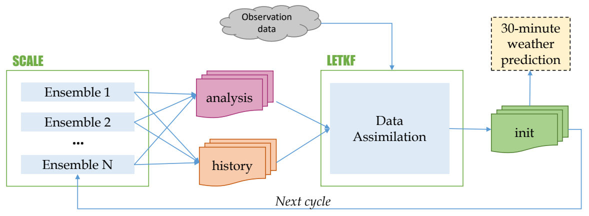

Fig. 5. SCALE-LETKF weather forecasting workflow.

Similar to other operational weather forecasting workflows, SCALE-LETKF consists of two components developed separately; a numerical weather prediction (NWP) model and a data assimilation system. The NWP model is the Scalable Computing for Advanced Library and Environment-LES (*SCALE-LES*), which simulates the time evolution of the weather-related atmosphere and land/sea surfaces based on physical equations. On the other hand, the data assimilation method uses the Local Ensemble Transform Kalman Filter (*LETKF*), which assimilates observation data taken from the real world into the simulated state to produce a better initial condition for the model. Figure 5 shows that the two components run in a cyclic fashion, exchanging data between the simulation and data assimilation phases in each cycle.

Additionally, observation data are streamed directly into RIKEN's supercomputing facility when executing the workflow in real-time. Although executed sequentially, there is potential for overlap between simulation and data assimilation, since the data assimilation processes perform a number of data transformation steps before proceeding to their main computation.

There are multiple ways how SCALE-LETKF can be deployed with respect to the placement of MPI processes on compute nodes. Due to the limitations imposed by the K Computer's job management system, the most common scenario has been to spawn SCALE and LETKF MPI jobs subsequently via separate `mpirun` invocations. Since SCALE and LETKF processes do not overlap in time, they currently communicate through the parallel file system. Another deployment scenario is to spawn SCALE and LETKF MPI jobs to separate sets of compute nodes and enable direct communication over the interconnect fabric [11]. In this configuration, MPI processes spawn multiple execution cycles of the workflow.

Challenges: Ideally, the SCALE and LETKF processes should be located on the same compute nodes to minimize inter-job communication costs. The main challenge is that this would require the batch job system and the runtime to provide mechanisms that enable flexible resource sharing between the two components. Specifically, the runtime system should provide standard methods for synchronization and node resource re-partitioning so that components could reserve and release CPU cores dynamically.
Issue: Resource affinity control of multi-component workflows.

F. Deep Learning in Cancer Problems

We are using the Exascale Computing Project (ECP) CANDLE application [12] with the Pilot 3 data set to illustrate the impact of conflicting directives from different parts of the software stack. CANDLE Pilot 3 is a multi-task, deep neural network (DNN) for data extraction from clinical reports. It uses TensorFlow and the Intel MKL-DNN deep learning library. The latter uses OpenMP to parallelize its work. OpenMP environment variables can be used to control the number and placement of the MKL threads. TensorFlow uses two thread pools, `intra_op` and `inter_op`, to stage work. The user can control the size of the thread pools, but not the placement.

The top line in Figure 6 shows the initial run with 15 trials in the recommended configuration using the mOS multi-kernel [13]. The gray data points below it show CANDLE running in the same configuration but under Linux. Because Linux exhibits such wide variation from run to run, we ran 100 trials of all experiments after the initial mOS runs.

When investigating why mOS was slower than Linux, we discovered that the recommended settings caused multiple worker threads to run simultaneously on the same CPUs. Doing a parameter sweep we were able to find a better combination of thread pool sizes and OpenMP thread placement. The results are shown in the lower two data sets in Figure 6. Linux improved by about 9% and mOS by 20%. We achieved these improvements doing a search over the parameter space and adjusting the configuration manually in setup scripts.

Challenges: A user of an application that uses multiple runtime layers with conflicting thread placements may not be aware that the application could perform much better, and what

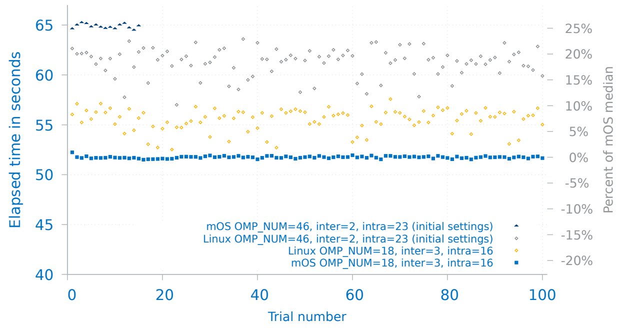

Fig. 6. CANDLE Pilot 3 runs with conflicting and optimized thread placement under two types of systems: monolithic (Linux) and multi-kernel (mOS).

the root cause of the problem is. Even when the problem has been identified, it is not obvious how to solve it. Each runtime layer uses different mechanisms to control thread placement. This limits what can be done. In the example above, the thread pool capacities that Tensorflow uses can be configured, but not the placement of the threads. Luckily, we were able to move the MKL OpenMP threads. A system where the components of an application could coordinate would let users discover earlier that there is a resource management conflict and make the optimizations easier and portable.

Issue: Multiple, uncoordinated types of threads.

III. Salient Themes and Requirements

The use cases above are real-world examples of application domains that require runtime management beyond what is available today, or accessible to non-advanced users. To design a framework for multi-runtime resource management in next-generation systems, we need to clearly understand the requirements of applications. In this section, we list the salient challenges we uncovered in the application use cases. Our goal is to replace application- and architecture-specific solutions with a unifying approach that benefits a broad class of applications and usage models. Table I cross-references our application examples with the themes and programming abstractions we observed.

A—Multiple, uncoordinated types of threads. A common theme across most use cases is the highly dynamic and heterogeneous nature of runtime components. Most applications have several phases as a result of multiple physics packages, for example, or the composition of multiple jobs linked together to pursue a common goal. These phases neither share the same complexity, nor the same behavior regarding parallelism. Each phase may use a specific parallel language/runtime with its own set of workers and its own set of resources.

We differentiate between *explicit* workers that are explicitly launched and managed by an application and *implicit* workers that are launched and managed by third-party libraries. We emphasize that applications may not have visibility into implicit workers. We further consider *compute* workers and *utility* workers. Utility workers are those used by utility libraries such as nonblocking communication threads or system services threads (see Section IV).

B—Dynamic work performed by auxiliary libraries. An experienced developer may be able to rebalance MPI and OpenMP workers based on when and how many OpenMP threads are active. Some of our use cases do this. However, this reconfiguration may not be portable or possible with workers that application developers are unaware of, i.e., implicit.

Both compute and utility workers may be considered implicit workers in auxiliary libraries. Implicit compute workers include threads from math and linear algebra packages such as Intel's MKL. Candle and LBANN include these type of workers. Utility workers, on the other hand, include nonblocking communication progress threads from the MPI library and the NVIDIA NCCL library for communication between GPUs. GeoFEM, NWCHEM, and LBANN (discussed in Sections II-A and II-D) are examples that include utility workers. This type of workers may have different requirements than compute workers. For example, performance may be affected by their distance to the network controller or to GPUs. Since application developers may not be aware of all the workers running on behalf of an application, it can be difficult to derive performant resource mapping policies.

C—Rebalancing and remapping of MPI processes and threads in different application phases. A prevalent hybrid programming model in HPC is the use of message passing for inter-node communication and shared memory for intra-node communication. Efficient data exchange among local threads requires careful placement and scheduling. Unfortunately, each runtime attempts to optimize this for its own workers without regard for other runtimes in the composed application stack.

The most common case of hybrid programming is MPI+OpenMP. A particular challenging configuration is pure MPI phases intermingled with MPI+OpenMP phases. Without dynamic reconfiguration of processes and threads between phases, some compute resources can be left idle while others are overloaded with multiple workers. Under- or over-subscribing compute resource can significantly limit scalability. Application use cases where dynamic rebalancing is required include SCRIP, Hydrodynamics, and SCALE+LETKF, discussed in Sections II-B, II-C, and II-E, respectively. Currently, runtimes lack the ability to share information with each other to be able to adapt their workers and their placement to leverage all of the available resources efficiently.

D—Multiple applications working together on the same problem. A challenging configuration for runtime coordination is multiple concurrent applications. This arises when multiple jobs, part of the same simulation, are launched concurrently and share resources. Examples include in-situ analytics and code coupling workflows such as SCRIP and SCALE-LETKF (see Sections II-B and II-E). Code coupling combines a set of application components, often developed separately, working together to achieve a common goal.

Since each component is launched separately with its own

TABLE I
TAXONOMY OF APPLICATIONS BASED ON THEMES A-D AND PARALLEL PROGRAMMING ABSTRACTIONS.

			Candle	GeoFEM	NWCHEM	SCRIP	Hydro	LBANN	SCALE+LETKF
A/B	Dynamic compute workers	Explicit		✓	✓	✓	✓	✓	✓
		Implicit	✓					✓	
B	Dynamic utility workers			✓	✓			✓	
C	Rebalancing and remapping					✓	✓		✓
D	Multiple applications	Single	✓	✓	✓		✓	✓	
		Multiple				✓			✓
Parallel programming abstractions		MPI	✓	✓	✓	✓	✓	✓	✓
		OpenMP	✓	✓	✓	✓	✓	✓	✓
		POSIX threads						✓	
		NVIDIA CUDA						✓	
		C++ threads	✓					✓	

resource allocation, its knowledge of resource availability is limited to its local view. The resource allocator is the only entity that keeps track of all resources allocated to the overall workflow. To have a global view and efficiently manage all resources, each local library would have to exchange information with the resource allocator. Only if they all have information about the workers and resources available in the system, can their local libraries make better mappings and notify each other about the new resource distribution.

Based on the application use cases and their resource management challenges, we outline a set of requirements for a general multi-layer resource management framework that can address the needs of emerging workflows and improve application productivity: (1) Dynamic runtime placement based on worker characteristics including compute and utility threads; (2) Coordination of all workers from all runtimes within a compute node; (3) Dynamic reconfiguration and remapping of workers including processes and threads; and (4) Worker management across multiple concurrent jobs.

IV. Proposed Approach

In this section, we propose a system framework to address the challenges posed by the applications described earlier. First, some terminology and assumptions:

- For any given user, the *global resource manager* (RM) grants hardware resources on the machine. These resources include multiple compute nodes and, within each node, compute cores, GPUs, memory, etc. We refer to these granted resources as a *user allocation*.
- Within a user allocation the user may launch one or more *jobs* (a parallel program or a composition of programs) sequentially or concurrently.
- A parallel program consists of one or more *tasks* or *processes* and each task may include multiple *threads*.
- A compute node executes jobs from one or more users as well as *system services*.
- System services include processes associated with the OS, parallel file system, RM, etc., and are often run on isolated resources. Well-known techniques to mitigate application interference from system services include Cray's core specialization [14] and Fujitsu's system cores [15].

A central component of the proposed framework is the *Mapping Coordinator* (MC), a cross-stack coordination layer in charge of mapping runtime components to the available hardware resources. Once resources are granted to a user by the global resource manager, the Mapping Coordinator coordinates within-node access to these resources. We emphasize this distinction, as the Mapping Coordinator is not intended to replace the resource manager. Instead, it provides the missing piece of coordination among multiple runtime components once resources have been assigned on a compute node. To this end, the Mapping coordinator provides user interfaces to request resources and, with this information, it arbitrates placement among runtime components based on resource availability. The Mapping Coordinator provides a set of building blocks for node resource coordination and management upon which more advanced optimization layers can be built such as end-to-end workflow managers.

Figure 7 demonstrates the role of the Mapping Coordinator. In this example, the resource manager has granted resources as follows: resources 0-3 to system services, resources 4-14 to Julie's jobs, and resources 15-20 to Nadine's job. The resource manager is in charge of isolating resources between users, as well as between users and system services on a compute node.

When Julie launches job 1 (consisting of processes P0 and P1), the Mapping Coordinator assigns each process onto a subset of the resources in Julie's allocation according to a mapping policy, the hardware topology, and available resources. If each process has three threads, for example, a mapping policy may place processes 0 and 1 on resources 4-6 and 7-9, respectively. Similarly, the Mapping Coordinator maps Julie's job 2 onto resources 10-14. If job 2 launches threads, the Mapping Coordinator will map them onto the scope of resources associated with this job. Since the Mapping Coordinator keeps track of all the work executed, it reduces contention and conflicts.

The Mapping Coordinator includes the following abstractions: *scopes* to abstract hardware resources associated with a specific execution context (Section IV-A); *affinity policies* to

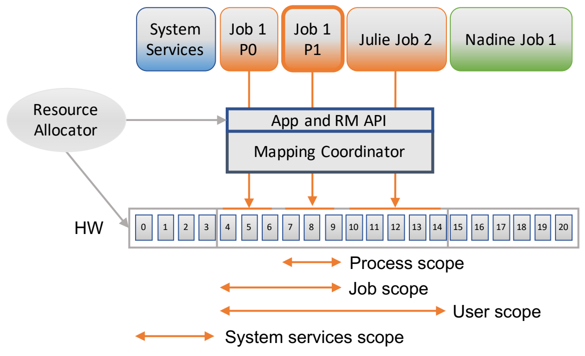

Fig. 7. The Mapping Coordinator, the central component of the proposed framework, orchestrates the placement of all workers onto the granted hardware resources. Scopes associated with Julie's job 1 process 1 are also shown. Hardware resources are depicted as numbered gray rectangles.

indicate how to map runtime components onto the hardware at a high-level (Section IV-B); and a *functional interface* for applications, libraries, and resource managers to interact with the MC (Section IV-C).

A. Resource Scopes

User libraries and applications can use *scopes* to help the Mapping Coordinator find the best placement of workers within the constraints of the resources associated with a user. A scope is an abstract representation of hardware resources that may include cores, memory, and accelerators. One may use scopes to express affinity in a more general way than, for example, CPUs or NUMA domains. Scopes are hierarchical, but not necessary disjoint at a given level.

For any given user, we associate four types of scopes: process, job, user, and system services. Figure 7 shows an example of the various scopes associated with Julie's job 1 process 1. The process scope is composed of those resources assigned to process 1 (7-9); the job scope is composed of resources 4-9, i.e., those assigned to job 1; the user scope is composed of resources 4-14, i.e., those assigned to Julie; and the system services scope is composed of resources 0-3, where the OS and other system services execute. In addition to these system-defined scopes provided by the Mapping Coordinator, users may create scopes dynamically.

B. Mapping and Binding Policies

An important part of the proposed framework is the ability to leverage and specify mapping policies that determine the way in which processes, threads, and GPU kernels map to the hardware. These mappings have a substantial impact on performance and, at the same time, can be very complex because of the heterogeneous nature of emerging systems.

We recognize that applications have different requirements and no single mapping policy can meet the demands of all applications. To this end, the proposed framework allows incorporating different affinity policies that can be applied dynamically at different granularities, including on a code-phase basis. What this means to an application developer is choosing mapping policy A or B rather than specifying cores, memory domains, and GPUs where processes, threads, and kernels should run. While standardized affinity policies, such as those specified in OpenMP 4 and above, are important, our goal is to enable and incorporate emerging policies that focus on optimizing applications based on heterogeneous aspects of a system such as accelerators and memory.

C. Main Functions of the Mapping Coordinator

Map and bind runtime components to hardware resources. The primary function of the Mapping Coordinator is to enable runtime components to map hardware resources efficiently.

Provide low-level interfaces as well as high-level mapping polices. To achieve the above mentioned goal we envision the MC providing interfaces at two levels of abstraction. The low-level interface allows arbitrary customization with respect to the association of resources to runtime components, while the high-level policies express intuitive, frequently used patterns on how mappings are established.

Keep track of resource utilization. In order to provide resource mappings, the Mapping Coordinator internally keeps track of resource utilization.

Provide an interface to query available resources. Runtime components may query the state of resource usage at any time.

Provide an interface to request and release resources. For components that require precise resource designation the MC provides interfaces to request and release specific resources.

Arbitrate access to resources to avoid or reduce oversubscription. The MC also serves as a per-user synchronization point on each compute node enabling runtime components to efficiently arbitrate resources among each other.

Provide an interface to dynamically rebalance processes and threads. To enable reconfiguration of resources among runtime components, the Mapping Coordinator provides an interface to reconfigure different types of workers and their hardware mapping dynamically. This would allow coupled simulations with heterogeneous kernels to inter-operate and utilize the best configuration for each kernel.

Provide an interface to notify of changes in the resource set associated with a user allocation. Resource managers are evolving to provide multi-tenancy and the ability to grow and shrink user allocations. For example, if a job from user A completes on a shared node with user B, the resource manger may reassign A's resources to user B. The MC needs awareness of dynamically changing allocations to provide not only efficient but valid mappings. Thus, the MC provides an interface allowing resource managers to publish when a user allocation has changed.

V. Related Work

There is numerous work related to the individual pieces of our proposed framework. Unlike other studies, our work places a strong emphasis on understanding the limitations of real applications, to help derive key requirements for next-generation systems. Below, we outline impactful studies that have helped shape our design of a general framework for coordination and arbitration across multiple runtimes.

We start by enumerating efforts addressing mapping and placement of processes and threads onto the hardware. The main limitation in this category is that most frameworks are specific to an MPI library implementation, vendor, or hardware architecture. In other words, portability is not a first-class concern. Furthermore, many of these affinity solutions require low-level hardware topology information, which makes it harder for application developers to use. Open MPI's LAMA [16] provided a rich set of options to enable user-defined affinity policies. MPIPP [17] is a placement framework that takes into account the characteristics of the target hardware, but does not address current architectures with complex, hierarchical multicore nodes. Other work in this area include PTRAM [18], TopoMapping [19], RAHTM [20], TreeMatch [21], LibTopoMap [22], EagerMap [23], and HierTopoMap [24]. Vendor solutions tailored to MPI implementations include those by IBM [25], Cray [26], and HP [27].

LIKWID [28] provides flexible binding of MPI+Threads applications, but also requires integration with specific MPI implementations. QUO [3], on the other hand, is an MPI implementation-agnostic library that allows for dynamic placement. The user can specify mapping policies that are dynamically enforced on parts of an MPI+Threads application, making QUO suitable for multi-kernel applications such as those represented by Figure 2.

We also note that OpenMP [29], unlike MPI, has primitives for affinity and binding defined into the standard. This represents a significant step toward reaching portability, at least from a library implementation point of view. Our approach embraces this type of standard interfaces, which can be applied within the Mapping Coordinator's scope abstraction.

Resource and Job Management Software (RJMS) provides task and job placement options. A few examples include Torque [30], OAR [31], and Flux [32]. Torque proposes NUMA-aware job placement. OAR provides a way to place application proceses on a flexible hierarchical representation of resources. This work takes into account the network topology, but the node architecture is left unaddressed. Flux provides hierarchical scheduling of resources to allow efficient placement of complex ensembles of jobs, and coordination among jobs in an ensemble. Coordination between different runtimes is not addressed, however. Kubernetes [33], a popular container management framework in cloud environments, is being considered in HPC as a possible replacement technology to job submission systems. Unlike Kubernetes, our approach orchestrates resource coordination at a much finer granularity than containers.

The Lithe system [34] provides a low-level interface for composing multiple runtimes and coordinating access to the hardware resources. Despite this compelling work, the problem of runtime composition still exists. Our work shares similar goals and, in addition, focuses on a high-level interface and abstractions to realize runtime composition at the application level. More general approaches to coordination and arbitration across multiple software entities include system designs for composing applications across operating systems and runtimes in multi-enclave HPC infrastructures, such as Hobbes [35] and Argo [36]. Multi-kernel operating systems such as mOS [13] and McKernel [37] inherently designate subsets of compute resources for specific needs. These systems, however, partition resources in a static fashion and do not typically address dynamic reconfiguration.

Finally, PMIx provides an API for exchanging information between components of the HPC stack, such as runtimes and the resource manager. PMIx has been proposed for resource coordination among containers [38] and is a candidate for implementing information exchange for arbitration in the proposed Mapping Coordinator.

VI. Summary

Scientific discovery is increasingly enabled by heterogeneous computing hardware. To utilize this hardware, scientists must compose their applications using a combination of programming models and runtime systems. Since these systems were designed in isolation, their concurrent execution results in resource contention and interference that limits application performance and productivity.

In this paper, we characterize this problem by analyzing seven real-world applications and quantify their limitations on current HPC software environments. We have drawn applications from various scientific fields and leverage them as use cases to identify the underlying functionality needed in next-generation systems. Understanding these requirements, derived from real applications, is a substantial step toward devising a productive software stack for next-generation HPC environments.

These use cases inform the general framework we are proposing to address resource contention and interference from multiple runtime systems. We focus on a cross-stack coordination layer called the *Mapping Coordinator*, which provides key functionality and interfaces to dynamically manage node-local resources based on user demand and resource availability. It satisfies user requests, minimizing interference and resource contention. We also describe high-level abstractions, including resource scopes and mapping policies, to help manage the complexity and portability challenges involved in worker affinity and placement. Finally, we are working on implementing the proposed framework with a particular emphasis on providing interfaces that can be used across operating systems, resource managers, and computer architectures. A detailed evaluation is the subject of future work.

Acknowledgment

We would like to thank Eric Maisonnave and Andrea Piacentini from CERFACS and Brian Van Essen from LLNL. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and the Exascale Computing Research laboratory with support of CEA, Intel, and UVSQ. LLNL-CONF-809181.

REFERENCES

[1] K. Nakajima, "Optimization of serial and parallel communications for parallel geometric multigrid method," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2014, pp. 25–32.

[2] A. Piacentini, E. Maisonnave, G. Jonville, L. Coquart, and S. Valcke, "A parallel SCRIP interpolation library for OASIS," CECI, UMR CERFACS/CNRS, France, Tech. Rep. WN/CMGC/18/34, 2018.

[3] S. K. Gutierrez, K. Davis, D. C. Arnold, R. S. Baker, R. W. Robey, P. S. McCormick, D. Holladay, J. A. Dahl, R. J. Zerr, F. Weik, and C. Junghans, "Accommodating thread-level heterogeneity in coupled parallel applications," in *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 2017, pp. 469–478.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[5] B. Van Essen, H. Kim, R. Pearce, K. Boakye, and B. Chen, "LBANN: Livermore big artificial neural network HPC toolkit," in *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, ser. MLHPC '15. New York, NY, USA: ACM, 2015, pp. 5:1–5:6.

[6] E. J. Bylaska, W. A. de Jong, N. Govind, and K. Kowalski, "NWChem, a computational chemistry package for parallel computers, version 4.5," Jan. 2007.

[7] A. Craig, S. Valcke, and L. Coquart, "Development and performance of a new version of the OASIS coupler, OASIS3-MCT_3.0," *Geoscientific Model Development*, vol. 10, no. 9, pp. 3297–3308, 2017.

[8] E. Maisonnave and S. Masson, "NEMO 4.0 performance: how to identify and reduce unnecessary communications," Sorbonne Universités-CNRS-IRD-MNHN, Paris, France, Tech. Rep. TR/CMGC/19/19, 2019.

[9] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. V. Essen, "Aluminum: An asynchronous, GPU-aware communication library optimized for large-scale training of deep neural networks on HPC systems," in *Workshop on Machine Learning in High-Performance Computing Environments*, ser. MLHPC'18, Nov. 2018.

[10] T. Miyoshi, G. Y. Lien, S. Satoh, T. Ushio, K. Bessho, H. Tomita, S. Nishizawa, R. Yoshida, S. A. Adachi, J. Liao, B. Gerofi, Y. Ishikawa, M. Kunii, J. Ruiz, Y. Maejima, S. Otsuka, M. Otsuka, K. Okamoto, and H. Seko, "Big data assimilation; toward post-petascale severe weather prediction: An overview and progress," *Proceedings of the IEEE*, vol. 104, no. 11, 2016.

[11] T. V. Martsinkevich, B. Gerofi, G.-Y. Lien, S. Nishizawa, W.-k. Liao, T. Miyoshi, H. Tomita, Y. Ishikawa, and A. Choudhary, "DTF: An I/O arbitration framework for multi-component data processing workflows," in *High Performance Computing*, ser. ISC'18. Springer International Publishing, 2018, pp. 63–80.

[12] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof, C. G. Cardona, B. V. Essen, and M. Baughman, "CANDLE/supervisor: a workflow framework for machine learning applied to cancer research," *BMC Bioinformatics*, vol. 19, no. 18, p. 491, Dec 2018.

[13] R. Riesen and R. W. Wisniewski, "mOS for HPC," in *Operating Systems for Supercomputers and High Performance Computing*, ser. High-Performance Computing, B. Gerofi, Y. Ishikawa, R. Riesen, and R. W. Wisniewski, Eds. Springer Singapore, Dec. 2019, ch. 18, pp. 307–334.

[14] L. Kaplan and J. Harrell, *Cray Compute Node Linux*, ser. High-Performance Computing. Springer Singapore, Dec. 2019, pp. 99–120.

[15] T. Kato and K. Hirai, *K Computer*, ser. High-Performance Computing. Springer Singapore, Dec. 2019, pp. 183–197.

[16] J. Hursey and J. M. Squyres, "Advancing application process affinity experimentation: open MPI's LAMA-based affinity interface," in *20th European MPI Users's Group Meeting, EuroMPI '13, Madrid, Spain - September 15 - 18, 2013*. ACM, 2013, pp. 163–168.

[17] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: An automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters," in *Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006*, G. K. Egan and Y. Muraoka, Eds. ACM, 2006, pp. 353–360.

[18] S. H. Mirsadeghi and A. Afsahi, "PTRAM: A parallel topology-and routing-aware mapping framework for large-scale HPC systems," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 386–396.

[19] J. J. Galvez, N. Jain, and L. V. Kalé, "Automatic topology mapping of diverse large-scale parallel applications," in *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*. ACM, 2017, pp. 17:1–17:10.

[20] A. H. Abdel-Gawad, M. Thottethodi, and A. Bhatele, "RAHTM: routing algorithm aware hierarchical task mapping," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*. IEEE Computer Society, 2014, pp. 325–335.

[21] E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters: Algorithmic issues and practical techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 993–1002, 2014.

[22] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011*, D. K. Lowenthal, B. R. de Supinski, and S. A. McKee, Eds., 2011, pp. 75–84.

[23] E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux, "EagerMap: A task mapping algorithm to improve communication and load balancing in clusters of multicore systems," *ACM Transactions on Parallel Computing*, vol. 5, no. 4, pp. 17:1–17:24, 2019.

[24] J. Wu, X. Xiong, and Z. Lan, "Hierarchical task mapping for parallel applications on supercomputers," *The Journal of Supercomputing*, vol. 71, no. 5, pp. 1776–1802, 2015.

[25] E. Duesterwald, R. W. Wisniewski, P. F. Sweeney, G. Cascaval, and S. E. Smith, "Method and system for optimizing communication in MPI programs for an execution environment," 2008. [Online]. Available: http://www.faqs.org/patents/app/20080288957

[26] Cray, "Cray performance measurement and analysis tool," 2017. [Online]. Available: https://pubs.cray.com/content/S-2376/7.0.0/cray-performance-measurement-and-analysis-tools-user-guide/about-the-cray-performance-measurement-and-analysis-tools-user-guide,

[27] D. Solt, "A profile based approach for topology aware MPI rank placement," 2007. [Online]. Available: http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt

[28] J. Treibig, G. Hager, and G. Wellein, "LIKWID: Lightweight performance tools," in *Competence in High Performance Computing 2010*, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 165–175.

[29] O. A. R. Board, "OpenMP application programming interface," Nov. 2018, version 5.0.

[30] A. Computing, "Torque resource manager." [Online]. Available: http://docs.adaptivecomputing.com/torque/6-0-0/Content/topics/torque/2-jobs/monitoringJobs.htm

[31] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard, "A batch scheduler with high level components," in *Cluster Computing and Grid 2005 (CCGrid05)*. Cardiff, United Kingdom: IEEE, 2005.

[32] D. H. Ahn, J. Garlick, M. Grondona, D. Lipari, B. Springmeyer, and M. Schulz, "Flux: A next-generation resource management framework for large HPC centers," in *43rd International Conference on Parallel Processing Workshops, ICPPW 2014, Minneapolis, MN, USA, September 9-12, 2014*, 2014, pp. 9–17.

[33] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016.

[34] H. Pan, B. Hindman, and K. Asanovic, "Composing parallel software efficiently with Lithe," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, B. G. Zorn and A. Aiken, Eds. ACM, 2010, pp. 376–387.

[35] B. Kocoloski, J. Lange, K. Pedretti, and R. Brightwell, "Hobbes: A multi-kernel infrastructure for application composition," in *Operating Systems for Supercomputers and High Performance Computing*,

B. Gerofi, Y. Ishikawa, R. Riesen, and R. W. Wisniewski, Eds. Singapore: Springer, Oct. 2019.

[36] S. Perarnau, R. Gupta, P. Beckman, P. Balaji, C. Bordage, G. Bosilca, F. Cappello, J. Dongarra, D. Ellsworth, B. V. Essen, D. Genet, R. Gioiosa, M. Gokhale, T. Herault, H. Hoffman, K. Iskra, L. Kale, G. Kestor, S. Krishnamoorthy, E. A. León, J. Lifflander, H. Lu, A. Malony, N. Mishra, K. Raffenetti, B. Rountree, M. Schulz, S. Seo, S. Shende, M. Snir, W. Spear, Y. Sun, R. Thakur, K. Yoshii, X. Zheng, H. Zhang, and J. Zounmevo, "ARGO: An exascale operating system and runtime," in *International Conference for High Performance Computing, Networking, Storage and Analysis; Research Poster*, ser. SC'15. Austin, TX: ACM/IEEE, Nov. 2015.

[37] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa, "On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 1041–1050.

[38] G. Vallee, C. E. A. Gutierrez, and C. Clerget, "On-node resource manager for containerized HPC workloads," in *Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC*, ser. CANOPIE-HPC'19. Denver, CO: IEEE/ACM, Nov. 2019.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

(1) Ran CANDLE with Pilot 3 data set on an Intel Platinum 8168 CPU @ 2.70GHz under Linux 4.14.134 and the mOS multi-kernel v0.7 using CentOS Linux 7.
(2) Ran NEMO-BENCH on two dual-socket Xeon E5-2698v4 CPU @ 2.2GHz under bullx scs with various configurations of MPI processes and OpenMP threads.
(3) Ran GeoFEM sol1 and sol7i on 32 compute nodes of the JCAHPC Oakforest-PACS system; sol7i uses non-blocking collective calls, which can be accelerated by MPI progress threads.
(4) Ran NWChem to calculate energy of 10 water molecules on 32 compute nodes, each with two Intel Xeon E5-2680 v2 CPU @ 2.80GHz, under RHEL Server 6.5.

ARTIFACT AVAILABILITY

Software Artifact Availability: There are no author-created software artifacts.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: https://doi.org/10.5281/zenodo.3880099
Artifact name: Experimental environment, scripts, and
↪  data
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: (1) Intel Platinum 8168 CPU @ 2.70GHz, dual sockets, 24 cores each. 196 GB DDR4, no accelerators; (2) Intel Xeon E5-2698v4 @ 2.2GHz, dual sockets, 20 cores each; (3) Oakforest PACS, Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz. Quadrant flat mode (MCDRAM exposed as NUMA node 1). 272 logical CPUs (Hyper-Threading enabled); (4) Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz 10 CPU cores (Hyper-Threading disabled) x 2 sockets.

Operating systems and versions: (1) Linux experiments: CentOS Linux 7 with 4.14.134 from kernel.org mOS experiments: mOS v0.7 from https://github.com/intel/mOS based on 4.14.134 Linux; (2) bullx SCS; (3) CentOS Linux release 7.6.1810 (Core). Linux kernel version 3.10.0-693.11.6.el7.x86_64; (4) Red Hat Enterprise Linux Server release 6.5. Linux kernel version 2.6.32-754.27.1.el6.x86_64.

Compilers and versions: (1) Intel Parallel Studio XE 2018.3.051. Python 3.7 using anaconda for package management; (2) Intel v16.1.150; (3) Intel icc (ICC) 19.0.1.144 20181018; (4) Intel icc (ICC) 17.0.3 20170404.

Applications and versions: (1) CANDLE from the CORAL-2 Deep Learning Suite: https://asc.llnl.gov/coral-2-benchmarks/; (2) BENCH configuration of NEMO 4.0; (3) GeoFEM versions sol1, sol7i; (4) NWChem version 6.6, revision 27746 (2015-10-20).

Libraries and versions: (1) Python 3.7 using anaconda for package management. Packages: candle intelpython3_core intel tensorflow pandas keras scikit-learn requests opencv tqdm matplotlib graphviz pydot; (2) Intel MPI v5.1.2.150; (3) Intel MPI 2018.3.222 (without progress threads). Intel MPI 2019.1.144 release_mt (with progress threads). Hook pthread_create and control thread placement to emulate UTI library binding behavior; (4) MVAPICH2 version 2.1 modified to integrate UTI library UTI library: git commit hash: 348ed82; prefix hwloc symbols with 'uti_'.

Key algorithms: (1) OpenMP thread allocation algorithm: KMP_BLOCKTIME=0. Initial: KMP_AFFINITY="granularity=fine,compact,1,0." Optimized: KMP_AFFINITY="granularity=fine,proclist=[10-23,34-47,58-71,82-95],explicit;" (2) Switch hybrid parallel configuration at runtime. This dynamic parallelism is controlled by HIPPO: two different configurations are defined and used for SCRIP and NEMO routines, respectively. NEMO-BENCH: 40 and 80 MPI tasks per node. SCRIP: 1 MPI task with 40 and 80 OpeMP threads per node; (3) Process/thread binding: 8 ranks / node, 8 OMP threads / rank. KMP_AFFINITY=compact. KMP_HW_SUBSET=1T. Progress threads bound to logical CPU cores 68,69,134,135,66,67,202,203; (4) Process/thread binding: 16 ranks / node bound to physical CPU cores 2-9 and 12-19, 1 OMP thread / rank. Progress threads bound to CPU cores 0,1,10,11.

Input datasets and versions: (1) CANDLE Pilot 3 data set from https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot3 using the P3B1 benchmark p3b1_baseline_keras2.py; (2) BENCH-1 degree resolution configuration; (3) N/A; (4) Cluster: Ten water molecules. Method: Partial-direct CCSD(T). Basis: cc-pvdz. Task (what to calculate): Energy.