Dynamic Adaptation Techniques and Opportunities to Improve HPC Runtimes

Mohammad Alaul Haque Monil,
Email: mmonil@cs.uoregon.edu, University of Oregon.

***Abstract*—Exascale, a new era of computing, is knocking at the door. Leaving behind the days of high frequency, single-core processors, the new paradigm of multicore/manycore processors in complex heterogeneous systems dominates today's HPC landscape. With the advent of accelerators and special-purpose processors alongside general processors, the role of high performance computing (HPC) runtime systems has become crucial to support different computing paradigms under one umbrella. On one hand, modern HPC runtime systems have introduced a rich set of abstractions for supporting different technologies and hiding details from the HPC application developers. On the other hand, the underlying runtime layer has been equipped with techniques to efficiently synchronize, communicate, and map work to compute resources. Modern runtime layers can also dynamically adapt to achieve better performance and reduce energy consumption. However, the capabilities of runtime systems vary widely. In this study, the spectrum of HPC runtime systems is explored and evolution, common and dynamic features, and open problems are discussed.**

***Index Terms*—Dynamic Adaptation, Runtime System, High Performance Computing**

I. Introduction

Exascale computing is an eventual reality for which the HPC community is preparing. Even though there have not been any exascale systems as of now (Fugaku tops the TOP 500 list with 415 Petaflops as of SC20 [1]) some applications achieved exascale performance with mixed-precision floating point operations. After the break down of Dennard scaling [2], multicore and manycore systems along with heterogeneous nodes are the main enabling technology needed to realize the dream of achieving exascale machines. Heterogeneous systems allow machine designers to battle the temperature barrier that limits increasing computation power in single-core processors [3], [4]. Heterogeneous systems with multiple GPUs and multi/manycore CPUs are the go-to solutions for high-performance systems [5]. Moreover, special-purpose processors such as tensor processing units, deep learning accelerators, and vision processors are on the rise and soon to become a part of HPC facilities.

On one hand, to increase the computational power chip manufacturers are developing multicore and manycore processors such as Intel Xeon and Xeon-phi processors, NVIDIA Volta GPUs, AMD's Radeon GPUs, etc. In addition, chip manufacturers are now housing a variety of processing units serving different types of computing needs on a single die in the form of shared memory heterogeneous systems [6]. While Intel's Ivy bridge [7] and AMD's fusion [8], [9] architectures were among the early systems combining two compute-capable processing units (i.e., CPUs and GPUs) under the same memory subsystem, later generations of integrated heterogeneous systems such as NVIDIA's Tegra Xavier have taken heterogeneity within the same chip to the extreme. Processing units with diverse instruction set architectures (ISAs) are present in nodes in supercomputers such as Summit, where IBM Power9 CPUs are connected to 6 NVIDIA V100 GPUs. Similarly in integrated systems such as NVIDIA Xavier, processing units with diverse instruction set architectures work together to accelerate kernels belonging to emerging application domains. Moreover, large-scale distributed memory systems with complex network structures and modern network interface cards adds to this complexity. To efficiently manage these systems, efficient runtime systems are needed. While this race toward superior computing capacity increases the complexity of systems, it unveils new challenges for HPC runtime systems to efficiently utilize the resources by providing an effective medium between the applications and the machines.

Since the beginning of parallel computing, the capabilities of runtime systems have been shaped by innovations in computer architectures. Starting from a simple messaging passing architecture to the newest heterogeneous architectures with accelerators, runtime systems adapt to provide abstraction and efficient utilization. Over the years, different runtime systems have emerged to address the needs of the HPC community. Early HPC runtimes started as bulk synchronous Message Passing Interfaces (MPI [10]), where heavyweight processes communicate with each other through messages to asynchronous multitasking runtime systems and the computation is decomposed into fine-grained units of work. However modern HPC runtimes come in a wide variety. Along with addressing both shared and distributed memory architectures, runtime systems employ different execution models for running units of work on different processors in heterogeneous systems. Runtime systems designed for task-based execution (OpenMP [11], HPX [12], Charm++ [13], etc.) operate by decomposing the total workload into sub-workloads. Runtime systems designed for heterogeneous systems (e.g., StarPU [14]) maintain processor-wise queues to effectively schedule the workload to increase utilization for the system or to meet the user demand. There are also runtime systems built for accelerators like GPUs (CUDA runtime by NVIDIA [15] and HIP [16] by AMD). Runtime systems for accelerators are designed to efficiently execute the workload due to the differences in throughput-oriented execution in GPUs and CPUs.

HPC runtime systems are active entities during the execution of an HPC application and capable of providing dynamic decisions. Dynamic decisions, such as scheduling computations to processors or load balancing among compute resources, increase performance and utilization of the system. Moreover, some runtime systems can provide energy-aware decisions so that energy consumption is reduced [17]. Because energy consumption is one of the biggest concerns for exascale machines [18], having such a feature in a runtime layer is desirable for future HPC ecosystems. In order to improve the overall performance of the systems, some runtime systems expose interfaces to collect data or dynamically tune the parameters and provide various configurations to guide the execution. These dynamic features of the runtime systems vary widely which limit the development of any standard technique that works universally. Some runtime systems provide hooks or callbacks for external performance tools (OMPT [19]) but many others do not.

The evolving ecosystem of HPC runtimes keeps providing abstractions at the cost of increasing responsibilities at different layers. With access to the application that is running and the underlying hardware, an HPC runtime positions itself as an active component that can take application- and hardware-aware decisions. By providing a way to interpret the relationship between application and hardware, runtime systems are capable of performing more dynamic decisions to improve performance and reduce energy consumption. For this reason, the features and dynamic decision capabilities of runtime systems need to be studied. Moreover, the evolution of the HPC runtimes needs to be understood to identify what drives the change in the HPC ecosystem. For this reason, this study is organized as follows.

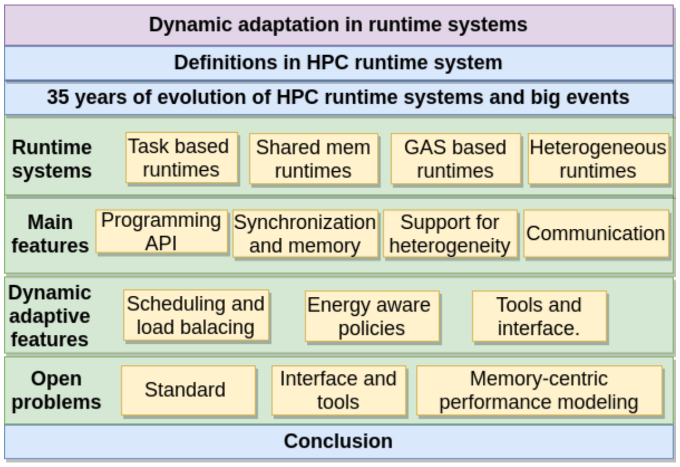

Dynamic adaptation in runtime systems
Definitions in HPC runtime system
35 years of evolution of HPC runtime systems and big events
Runtime systems
Task based runtimes
Shared mem runtimes
GAS based runtimes
Heterogeneous runtimes
Main features
Programming API
Synchronization and memory
Support for heterogeneity
Communication
Dynamic adaptive features
Scheduling and load balacing
Energy aware policies
Tools and interface.
Open problems
Standard
Interface and tools
Memory-centric performance modeling
Conclusion

Fig. 1: Organization of this study

A. *Organization*

There are three components of this study: 1) runtime systems, 2) dynamic adaptation techniques and capabilities of the runtime systems, and 3) opportunities for dynamic adaption. Having these three goals, this study is organized as Fig. 1. First, the definition of different terms and runtime systems are provided. Next, in order to understand the driving force behind the evolution of HPC runtimes, major events of the last 35 years are studied. This evolution study also helps to identify the state-of-the-art HPC runtimes of current times. Then, HPC runtimes are categorized. Each runtime system is then briefly studied to understand the concepts. The main features of the runtime systems are then compared to identify similarities and dissimilarities. Dynamic adaptation capabilities and features of the runtime systems are then studied. Finally, based on the trend of the runtime systems and observation acquired from 35 years, opportunities that would increase the dynamic adaption capabilities of the runtime systems are identified.

II. Definition of Runtime Systems

In order to properly define a runtime system, programming models, execution models, APIs, libraries, language extensions, and languages are discussed first.

A. *Programming models and execution models*

Both the programming model and execution model are logical concepts. The programming model is related to how a programmer designs the source code to perform a particular task to take advantage of the underlying runtime system and the hardware [20]. On the other hand, the execution model decides how a program written by following a particular programming model is executed in real time [21]. In short, the programming model is the style that is followed to program and the execution model is how the program executes. For example, the programming model of OpenMP defines how to express parallelism using directives and parallel regions, and the execution model refers to the multi-threaded execution of the code.

B. *Parallel programming APIs, libraries, language extensions, and languages*

Parallel programming APIs are a collection of entities designed to express the programming model and execution model in code. An API can be a new language, a language extension, or a collection of libraries. For example, the OpenMP [22] programming API is a language extension while the API for HPX [23] is based on libraries. Every programming language provides an execution model and enables the use of different parallel execution model at runtime.

C. *HPC runtimes*

A runtime system that defines the execution environment is an interface between the Operating System (OS) and the application. It abstracts the complexity of an OS and ensures portability to different OSs [24]. The execution model of a program is written by following a programming model through an API and is ensured by the runtime system. For example, the runtime system of the C language provides memory management. A C compiler inserts instructions into the executable and when it runs, the runtime system, which is a part of the executable, manages the stack to provide all the memory management functionalities. However, HPC runtimes go far beyond the capabilities provided by the basic languages.

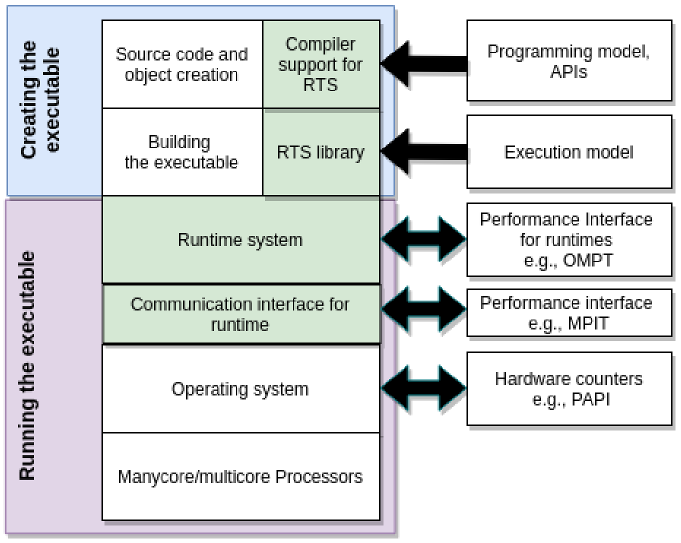
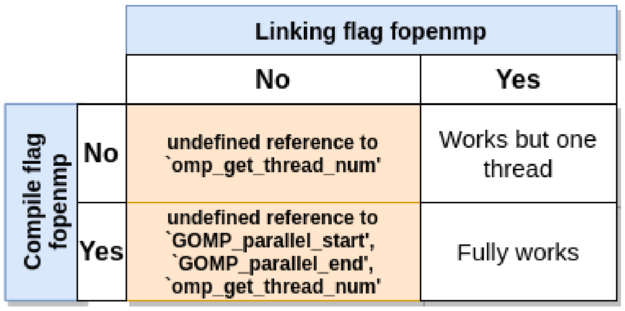

Creating the executable
Source code and object creation
Compiler support for RTS
Programming model, APIs
Building the executable
RTS library
Execution model
Running the executable
Runtime system
Performance Interface for runtimes e.g., OMPT
Communication interface for runtime
Performance interface e.g., MPIT
Operating system
Hardware counters e.g., PAPI
Manycore/multicore Processors

Fig. 2: Runtime system



		Linking flag fopenmp	
		No	Yes
Compile flag fopenmp	No	undefined reference to `omp_get_thread_num'	Works but one thread
	Yes	undefined reference to `GOMP_parallel_start', `GOMP_parallel_end', `omp_get_thread_num'	Fully works

Fig. 3: Example of OpenMP hello-world

An HPC runtime system becomes an active entity that is capable of making dynamic decisions based on the execution model and in some cases, performs communication while the executable is running. Some programming languages in the HPC domain host powerful runtime systems. For example, Chapel [25] is a Partitioned Global Address Space (PGAS or GAS) language; however, its runtime system manages a global address space. For this reason, some HPC languages are also considered in this study.

Figure 2 shows a layer-wise diagram of a runtime system. At the top layer, the programming model is expressed through source code by calling the API. When the object file is created, the compiler inserts necessary codes to carry out necessary runtime activities. In the second layer, when the executable is built by resolving all the library calls, the execution model is expressed (partially). When the program is executed, the runtime system at layer three sits on top of the communication layer (or the operating system) to reinforce the execution model by providing all the runtime facilities such as scheduling, load balancing, collecting data, etc. Since most of the HPC runtime systems are active components, there are interfaces, tools, or techniques available for direct interaction with the runtime layers. Using these interfaces, the activities of runtime systems can be monitored, altered or controlled. At level five, the operating system sits where hardware counter data can be collected. Some runtime systems collect the hardware counters from the operating system.

D. Example: how programming models and execution models work together

To demonstrate and clarify the relationship between a programming model and an execution model (a relationship that enables the runtime system to orchestrate the execution), a simple OpenMP experiment is presented. In this experiment, an OpenMP parallel region is declared to print the thread numbers. Figure 3 shows the impact when the compile and linking flag of GOMP (fopenmp) are used and omitted. When both the compile and linking flag are used, the code fully works and shows ids of different threads. In this case, the relationship between the programming model and the execution model is perfectly established and the runtime system can do its job properly. When no flag is used for both compilation and linking, the linker provides a linking error since it could not resolve the OpenMP API call. However, an interesting thing is observed when the compilation flag is not used but the linking flag is used. The exact code that is written by following the programming model and correct API calls only showed the execution of one thread. This happened because the compiler could not provide the necessary information to the runtime system to follow the execution model. This becomes more visible when the compilation flag is used but the linking flag is not. In this case, two additional linking errors showed up. $GOMP\_parallel\_start()$ and $GOMP\_parllel\_stop()$ are two function calls inserted by the compiler that enable the compiler to specify the execution model to the runtime system.

III. Evolution of HPC Runtimes

In this section, the evolution of HPC runtimes is discussed. In order to understand what shaped contemporary HPC runtimes, major events and runtimes developed over the last 35 years are explored. The reason behind choosing 1985 as the starting year because parallel computers started becoming more available around that time. It is no secret that HPC runtimes are majorly shaped by innovations in the field of computer architecture. A closer look is taken at the correlation between the ever-changing computer architectures and the evolution in runtime systems. The whole period from 1985 to the present day is divided into four parts considering the events in 10 years duration. After, the state-of-the-art runtimes are identified to take a closer look.

A. Before 1985

In the 1970s, for providing the highest performance, vector computing was the go-to solution [26] (such as single-processor machine Cray-1 [27] and multiprocessor machine Illiac [28]). In the 1980s, multiprocessor vector machines started becoming available in the form of Massively Parallel Processors (MPPs) (e.g. the concept of a hypercube [28])). Such an example is Caltech's Cosmic Cube [29] (fully operational

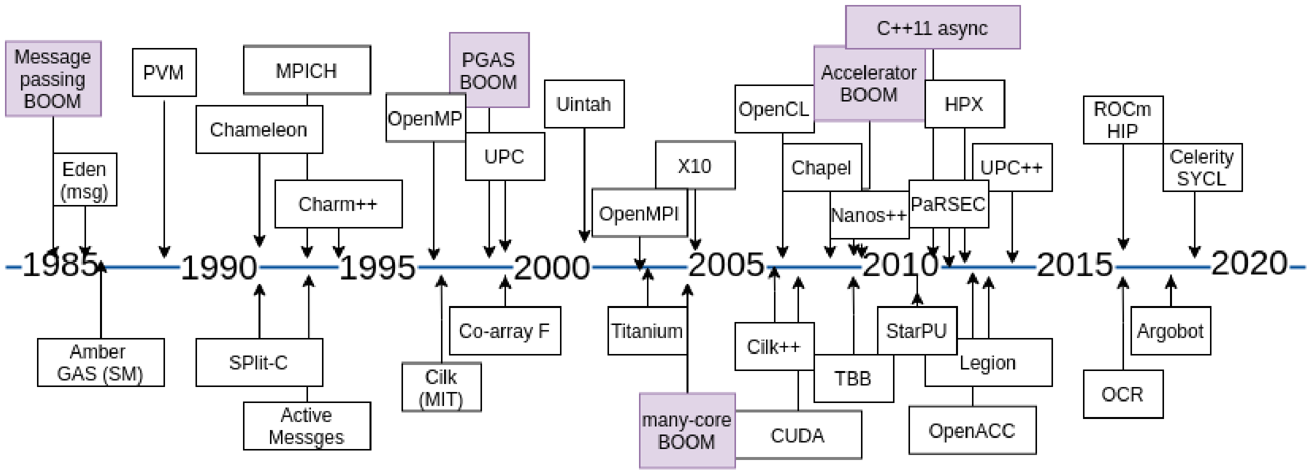


Fig. 4: Evolution of HPC runtimes for the last 35 years

in 1983) which was built using 64 nodes in point-to-point connection without shared memory. With this configuration message passing between nodes was necessary. Even though the idea of messaging passing architecture is successfully demonstrated by Cosmic Cube, the idea of message passing for distributed memory system already existed in other efforts, such as the Eden project for object style programming for distributed systems [30]. After the success of Cosmic Cube, other manufacturers started building cube machines and the trend lasted for a decade. At the same time, cluster computing, where compute nodes were loosely connected through network interface cards, was also gaining popularity in the 1980s for its simplicity in design [31]. Even though there were differences in architectures between hypercubes and clusters, during this decade it was established that a message passing architecture could provide high scalability at a low cost.

B. HPC runtimes in 1985-1995: message passing architecture

Having realized the potential of message passing architectures, vendors, national labs, and academia started providing programming interfaces that could facilitate message passing for distributed memory systems. Intel developed the NX/2 operating system [32] which provided messaging passing interface through system calls. Parasoft developed the Express [33] library that made a program portable to different machines (supported C and Fortran). Argonne National Laboratory (ANL) introduced p4 [34]. Similarly, IBM introduced the Venus [35] communication library for message passing. Moreover, PVM [36], a collaboration between Oak Ridge national laboratory (ORNL), University of Tennessee, and Emory University supported message passing for heterogeneous parallel computing. Chameleon [37] from ANL provided interoperability between different message passing libraries. These efforts focused on enabling message passing for parallel computing and took place in the late 1980s and early 1990s. However, because of different implementations by different manufacturers, portability became an important aspect. For this reason, in 1992, 60 people from 40 organizations started the Message Passing Interface (MPI) standardization effort at a workshop called “Workshop on Standards for Message Passing in a Distributed Memory Environment”. By 1993, the standard was completed and presented at SC93. In hindsight, this probably was one of the biggest initiatives that shaped modern high performance computing domain. After the standardization, MPICH [10] was released. MPICH (CH comes from Chameleon) is still one of the popular MPI implementations today. Even in 2021, MPI is the de-facto standard for HPC and it will not be an overstatement to say “MPI is everywhere”.

Apart from the rise of MPI, some other events took place which did not immediately become as successful as MPI, but planted the seed for the future. One such effort is the C language extension Split-C [38], which enabled the idea of a global address space by providing the option for declaring distributed array. With this configuration, one processor could reference pointers to another processor through communication. Even though the idea of a global address space was not new at that time, (Amber systems [39] mentions a similar idea about global address spaces) Split-C is considered to be the precursor of UPC, a PGAS language of modern times (SHMEM is also credited for one-sided communication which is one of the ideas of PGAS model [40]). Moreover, Charm++ [13], introduced in 1993, implemented a task programming model and is a precursor to the asynchronous many tasks (AMT) based runtimes. Another work, Active Messages [41], provided the idea of efficient message driven computation (opposed to message passing) which is found in modern HPC runtimes. The idea of loose synchronization is mentioned by Midway [42], which provided an opportunity to synchronize caches in distributed memory systems. The runtime carried out the synchronization through a barrier that is expressed by the programmer. The main idea was to have loosely bound synchronization criteria where the programmer was in charge of deciding whether a synchronization was necessary or not.

C. HPC runtimes in 1996-2005: shared memory and distributed shared memory

MPI standardization effort provided a great example of how a community-led effort could streamline an HPC Programming model. Riding off the success of MPI in distributed memory, a standard for shared memory programming models was introduced in 1997. First for Fortran and then for C, the OpenMP architecture review board (OARB) [11] released its standard to provide an alternative to MPI for increasing parallelism in shared memory systems. The standardization provided portability for OpenMP and was widely accepted by industry and academia. With OpenMP and MPI standardized, the HPC community was given two means to program both shared memory and distributed memory systems. By the end of the 1990s, the HPC community realized the drawbacks of explicit synchronous message passing at the front end for data transfer. For this reason, the concept of distributed shared memory gained traction and the concept of Partitioned Global Addressed Spaces (PGAS) was introduced. Three PGAS languages were launched in the late 1990s and early 2000s. UPC for C [43], Co-array Fortran for Fortran [44] and Titanium for Java [45]. These separate languages provided options to declare distributed arrays where the runtime system for the languages performed one-sided communication through communication interfaces like the GASNet communication library [46]. The GASNet communication interface is capable of using MPI for its one-sided communication. Later two more prominent PGAS languages were introduced, X10 [47] and Chapel [48], which introduced an asynchronous task model for distributed execution while utilizing a global address space.

D. HPC runtimes in 2005-2015: multicore, manycore and heterogeneity

In the mid-2000s, processor manufacturers were hit with the temperature barrier, which limited them from increasing processor performance by increasing the frequency in a single core. This obstacle led to the start of the manycore boom. Even though manycore processors existed previously, after 2005 they became mainstream for all levels of computing. This opportunity was realized by NVIDIA, a graphics processor manufacturer, who started manufacturing their graphics processor for general-purpose computing. This brought about the era GPGPUs. With the release of CUDA [49] by NVIDIA in 2007, accelerator programming drew the attention of the HPC community. As a result, runtime system providers started working towards supporting GPUs in their programming environments. Heterogeneous systems, where CPUs and GPUs are housed in a single node, led runtime developers to invent new approaches to harness the computing power from such systems. Programming standards and APIs for heterogeneous systems such as OpenCL [50], OpenACC [51] and StarPU [52] were introduced. These two major changes (multicore CPUs and manycore GPUs) increased in-node parallelism and exposed the disadvantages of using synchronisation-based message passing in MPI. Since explicit message passing with bulk synchronous barriers make both sender and receiver wait, it restricts the utilization of processor cores to reach their peak utilization (later MPI started providing asynchronous communication). For this reason, fine-grain tasks (instead of heavy MPI ranks or OpenMP threads) became one the most active fields in runtime systems research. Since lightweight tasks can be easily yielded and resumed when compared to heavy-weight OS threads, the asynchronous task based execution model received attention for increasing utilization of multicore processors. To provide high computation and communication overlap, many Asynchronous Many Task (AMT) runtime systems appeared, such as HPX [12], Cilk Plus [53], TBB [54], Legion [55], etc. Moreover, with C++11 released, highly templated code with improved asynchronous features began to be adopted by the runtimes. The AMT execution model is considered by the community to be a better fit for exascale computing, which is envisioned to appear by the early 2020s.

E. HPC runtimes in 2015-Present: asynchronous many task and abstraction

After 2015, the HPC community started focusing more on abstraction since multiple computing paradigms (CPU and GPU) in one node became common. For this reason, initiatives for providing programming approaches in a portable way (such as Kokkos [56]) started appearing. Moreover, AMD released its open-source GPU programming capability ROCm platform (HIP) [57]. Since there were already a considerable number of AMT runtimes introduced, the HPC community also started realizing the need for an AMT interface. A group of runtime system researchers from industry, national labs, and academia launched Open Community Runtime (OCR) [58] by releasing its specification. Moreover, Argobots [59] was introduced to work with different asynchronous many task execution runtimes.

F. Reduction and identification of state of the art

At this point, it is clear that there are a several domains into which HPC runtimes can be divided. Even though the MPI+X model is the commonly adopted model for scientific applications nowadays, many of the modern runtime systems use MPI at a communication level where MPI message sending and receiving is done by the runtimes rather than the programmer. Considering the recent developments, HPC runtimes can be categorized into four categories: 1. shared memory runtimes, 2. task based runtimes, 3. GAS based runtimes (languages), and 4. heterogeneous runtimes. Figure 5 shows the distribution of HPC runtimes. It is easy to notice that 20 runtime systems are chosen that represent the whole HPC runtime spectrum. HPC runtime survey papers [60], [61] are consulted to choose these runtimes. Some runtimes can provide multiple features, but the categorization is based on what the runtime is commonly known for. In order to show the overlap in the features provided by the runtimes, Table I shows which runtimes support which features. In this table, "Main" indicates the main objective of the runtime systems, "Extension" indicates later adoption, and "Supports" indicates

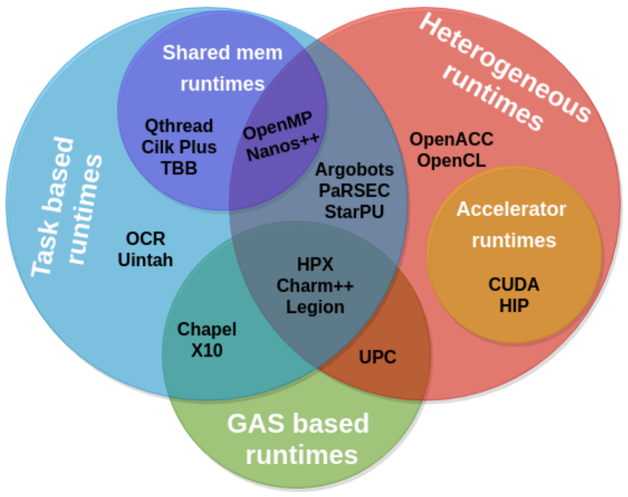


Fig. 5: Different state-of-the-art HPC runtime systems. White Text = Category name and Black Text = Runtime name. Runtime systems' color coding, Blue = Many task runtime, Green = GAS based runtime, Red = Heterogeneous capability enabler runtimes (Accelerator runtime included in this category), and Purple = Shared memory runtime. **This color coding is followed throughout the paper.** Note: MPI is not here. Because it's the top view. MPI is now part of many of the runtime systems.

the runtime is capable of supporting the feature, but that is not the main target for the runtime system.

Name	Shared Mem	Aync. MT	GAS	Heterogeneity
Cilk Plus	Main	Main	No	No
TBB	Main	Main	No	No
OpenMP	Main	Extension	No	Extension
Nanos++	Main	Main	No	Extension
Qthread	Main	Main	No	No
Charm++	Supports	Main	Main	Extension
HPX	Supports	Main	Main	Extension
Legion	Supports	Main	Main	Main
OCR	Supports	Main	No	Main
Argobots	Supports	Main	No	Main
Uintah	Supports	Main	No	Extension
PaRSEC	Supports	Main	No	Extension
UPC	Supports	Extension	Main	Extension
Chapel	Supports	Main	Main	Supports
X10	Supports	Main	Main	Extension
StarPU	Main	Main	No	Main
OpenCL	Main	No	No	Main
OpenACC	Main	No	No	Main
CUDA	Main	No	No	Main
HIP	Main	No	No	Main

TABLE I: Runtime systems categorisation. Main = main feature, Extension = extended later on, Supports = provide supports, and No = Does not support.

IV. Shared Memory Runtime Systems

In this section, shared memory runtime systems are briefly discussed. Table II lists the shared memory runtime systems.

A. *Cilk plus*

Cilk [62] originated from MIT in the mid-1990s and later Intel acquired it when MIT licensed it to Cilk Arts [63]. Intel released Cilk Plus [53], [64] as a part of the ICC compiler suite. Cilk plus uses a nonblocking spawn function to generate new tasks in a DAG which later syncs (spawn-sync), which implements the fork-join model. Cilk plus extended C/C++ by adding three keywords cilk_for, cilk_spawn, and cilk_sync. Cilk Plus provides a compiler-driven approach for task-level parallelism in shared memory machines. When tasks are spawned, they become a part of a DAG where data dependency exists. The scheduler enables the execution of the DAG by placing these tasks into OS threads when the tasks are ready to be executed. Cilk plus was supported by the Intel, GCC, and CLANG compilers. However, recently Intel decided not to continue support for Cilk Plus in the future releases.

B. *TBB*

OS thread-based solutions for programming multi-core systems are not portable. For this reason Intel TBB [54], [65] (oneTBB) is a C++ template library for threading abstraction. TBB is mainly designed for shared memory multicore CPUs. It expresses the parallelism in terms of logical tasks (C++ objects) which are scheduled to a pool of OS threads. In other words, it provides a wrapper to use the OS threads to make the program portable. The logical task provides a faster way to create work for OS threads, since creating and terminating OS threads are expensive operations (TBB tasks are 18 times faster than OS thread creation and termination). TBB provides a range of tools for parallel algorithms, including for loop, for-each loop, scan, reduce, pipeline, and sort implementations. These parallel constructs can be applied to various data structures such as queues, hash maps, unordered maps and sets, and vectors. TBB also supports both blocking and Lamba-based continuation-passing styles.

C. *OpenMP*

OpenMP [66] is one of the most popular and widely used names in the HPC community for its shared memory programming model. It is managed by OpenMP Architecture Review Board (OARB) [67]. This board has members from all leading manufacturers. OARB published the first specification [22] in 1997 for Fortran and in the following year, a C/C++ standard was released. The increasing availability of cache-coherent scalable shared memory multiprocessors (SSMPs, now called SMPs) in the mid-1990s inspired the creation of such a model. One of the main reasons the HPC community depended on MPI was the lack of cache coherence in the early machines (MPI's popularity for distributed memory systems is another reason). However, MPI was unable to provide increasing parallelism, which propelled the creation of OpenMP. In the beginning, OpenMP was targeted for data-parallel execution for CPUs only in Single Instruction Multiple Data (SIMD) fashion. In 2007, the task parallelism construct was proposed to increase the usability for OpenMP 3.0 specification and adopted the popular concept [68]. Later in 2013, OpenMP

4.0 specification included computation offloading options for GPUs. Using directives, parallel regions are declared. There is one master thread that forks multiple threads for data and task parallel computation. When a computation finishes, all the threads are joined to the master thread. For this reason, OpenMP is often referred to as a fork-join model.

D. OmpSs and Nanos++

OmpSs [69], [70] is an effort from Barcelona Supercomputing Center (BSC) which made an appearance in the HPC world in 2011. The main idea of OmpSs is to extend OpenMP and StarSs [71] for a directive-based asynchronous task execution model that also supports accelerators such as GPUs, FPGAs, etc. along with CPUs. OmpSs is implemented as an extension to OpenMP that enables asynchronous task features that target newer architectures like GPU, FPGA, etc. Over the years, many features from OmpSs were included in OpenMP specification [70]. (Similar to the relationship between C++ standard and Boost libraries). For this reason, OmpSs is considered as a forerunner of accelerator-based OpenMP. Like OpenMP, OmpSs also supports distributed computing through MPI. In OmpSs, provides data access directionality clauses (in, out, inout) to provide node level asynchronous execution by expressing data dependencies [72]. It also supports a single source file approach for accelerators. OmpSs is built on top of Mercurium source to source compiler [73] and the Nanos++ runtime system [74]. The Nanos++ runtime provides user-level task parallelization based on data-dependencies.

E. Qthreads

Qthreads [75], [76], an effort from Sandia lab introduced in 2008, is a user-level library for on-node multithreading. The initial target was to provide massive level multithreading with rich synchronisation [77]. With Qthreads, when an application exposes parallelism (specified by the user) in a massive number of lightweight user-level threads, the runtime system dynamically manages the scheduling of tasks. Apart from supporting a standalone execution option, Qthreads also used as a backend runtime for Chapel language [48]. Moreover, the Kokkos C++ library, a performance portability initiative from Sandia Lab for efficient management of data layout and parallelism for manycore processors is also extended to work with the Qthreads runtime [78]. For memory level synchronization, Qthreads uses the Full/Empty bit concept. In Qthreads, several shepherds are created during initialization. Shepherds are collections of user-level threads used to express memory regions (NUMA), processors, etc. A thread can be a part of multiple shepherds. The Qthreads API can limit the number of threads being created through futures, which is a user-level thread yet to be executed.

V. Task Based Runtime Systems

Task based runtime systems are discussed in this section. Table III shows task based runtimes.

A. Charm++

Charm++ [13] is one of the pioneers of modern asynchronous task based runtimes. It originated at the University of Illinois at Urbana Champaign (UIUC) in 1993. The Charm++ programming model and runtime implements a message-driven paradigm where computation starts after receiving messages. It works through parallel processes called *chares* which are C++ objects. These objects have entry points which are executed when a message is received. A program is over decomposed in terms of *chares* and the execution is completely non-deterministic since *chares* are invoked asynchronously [79]. The runtime maintains queues for ready messages and distributes them to processing elements for execution. Charm++ also maintains a global address space. For the last 30 years, Charm++ has matured and is one of the well-researched runtime systems in the HPC domain. It provides compatibility with new hardware, accelerators, and network technologies.

B. HPX

HPX [12] runtime is from Louisiana State University (LSU) and was introduced in 2014. HPX implements the concepts of the ParalleX execution model [80]. HPX strictly conforms to C++ standards and enables wait-free asynchronous execution. HPX implements active messages where computation is sent to data instead of sending data towards computation. In HPX, active messages are called *parcels* and processing elements are called *localities*. The runtime system implements an Active Global Address Space (AGAS) that is capable of object migration. AGAS generates the Global ID and GIDs that are used to locate an object in the system. Unlike systems such as X10 [81], which are based on PGAS, AGAS systems use a dynamic and adaptive address space that evolves over the lifetime of an application. Like Charm++, HPX is also built on the idea of over-decomposition of the work. The asynchronous nature of the execution ensures the compute resources are highly utilized since computation and communication are overlapped.

Name	Who	When	Open Source	Implementation	Target	One line description
Cilk Plus	MIT, Intel	1995	Yes	Extension	CPU	Spawn-join with non blocking task creation.
TBB	Intel	2007	Yes	Library	CPU	Light task/threads on OS threads with work-stealing scheduling.
OpenMP	OARB	1997	Yes	Extension	CPU and GPU	Directive based shared memory programming.
Nanos++	BSC	2011	Yes	Extension	CPU and GPU	Forerunner of OpenMP for asynchronous execution.
Qthread	Sandia	2008	Yes	Library	CPU	Back-end runtime for shared memory many tasks.

TABLE II: Shared memory runtime systems

Name	Who	When	Open Source	Implementation	Target	One line description
Charm++	UIUC	1993	Yes	Extension	Distributed	Pioneer of many task massage driven computation.
HPX	LSU	2014	Yes	Library	Distributed	Moving data to work and active GAS.
Legion	Stanford	2012	Yes	Library	Distributed	Logical region with GAS.
OCR	Intel, Rice U.	2016	Yes	Library	Distributed	An open community runtime interface.
Argobots	ANL	2016	Yes	Extension	Distributed	lightweight low-level threading API with stacking schedulers.
Uintah	Univ. of Utah	2000	Yes	Library	Distributed	Concurrent CPU-GPU scheduler for asynchronous execution.
PaRSEC	Utk Knoville	2013	Yes	Library	Distributed	Uses data-flow model in distributed heterogeneous environment.

TABLE III: Asynchronous many task runtime systems

C. Legion

Legion [82], [83] is an effort from Stanford University and Los Alamos National Laboratory (LANL). Legion is a data-centric programming model targeted for heterogeneous distributed systems. Legion aims to provide locality (data close to the computation) and independence (computation on disjoint data and can be placed on any compute component of the system). The main idea of Legion is based on three abstractions for data partitioning: using logical regions, a tree of tasks for using the regions, and a mapping interface for the underlying hardware. Logical regions are described by index spaces of rows and field space of columns. Each region can be sub-divided into sub-regions through index spaces (rows) and fields of spaces (columns). A tree of tasks starting from the root level with tasks spawning recursively into sub-tasks has specific access to logical regions. All tasks must specify which logical regions they have access to. A tree of tasks with its associated logical regions is mapped to different processing components in a distributed environment using the mapping interface of Legion. The mapping interface assigns the logical regions into physical memory. While the mapping decision can impact performance, it ultimately provides correct results. The mapping can be tuned for specific hardware because a mapping decision that can be best in one hardware may not be the best for other systems. While the correctness provides portability, the tuning is done by the users for better performance. Both the partitioning and mapping to hardware is the user's responsibility, while the runtime ensures coherence of data system-wide. Legion provides communication through another low-level runtime system called Realm [84] which supports asynchronous, an event-based runtime for task-based computations. Through Realm, Legion provides a global view of memory and support for CUDA. Realm is dependent on GASNet for communication through RDMA.

D. OCR

A comparatively new runtime system, the Open Community Runtime (OCR) [85] is a joint work from Intel and Rice University. Currently the University of Vienna [86] and PNNL have implementations of OCR, which are called OCR-Vx [86] and P-OCR [58], respectively. The main target of the runtime is to realize the opportunity of exascale systems. The authors argue that, in the exascale era, the HPC community will look for an alternative to MPI+X model. OCR started with its formal specifications [87]. The interface specification is similar to the MPI and OpenMP specification.s OCR is an asynchronous many task (AMT) runtime system for exascale where the main idea is to express computations through tasks, events, and data blocks. The data blocks are part of the application which is managed by the runtime. In a DAG of tasks, events are used to set up and resolve a dependency. OCR terms these tasks as event-driven tasks (EDTs). EDTs can run asynchronously and the runtime system guarantees when all the dependencies are met, EDT will run in the future. The tasks, events, and data blocks are presented as an object with pre and post slots for expressing the dependencies.

E. Argobots

Argobots is a lightweight low-level threading API developed at Argonne National Laboratory (ANL) as part of the project Argo in 2016 [59], [88]. Argobots provides integrated support for MPI, OpenMP, and I/O services. Argobots provides richer capabilities when compared to existing runtimes, offering more efficient interoperability than production OpenMP, a lower synchronization cost when MPI is used, and better I/O services. In Argobots, functions are expressed as ULT (ultra-light tasks) and tasklets. ULTs have a stack (similar to OS threads but smaller) which provides faster context-switching. ULTs can be yielded for dependency. On the other hand, tasklets are smaller and they use OS thread's memory space. Tasklets can not be yielded. These two levels of control beyond OS threads provide more options to express the parallelism in Argobots. Work units are expressed through creation, join, yield, yield_to, migration, and synchronizations primitives. Argobots developers are working to support Charm++ [89], OmpSs [70], Cilk [63], OpenMP [11] (BOLT [90]), XcalabeMP [91], and PaRSEC [92].

F. Unitah

Uintah [93]–[95] is a set of libraries for large-scale simulation. It provides a unified heterogeneous task scheduler and runtime which originated from the University of Utah's Imaging institute. Originally, Uintah supported an MPI-only approach for out-of-order execution. However, when multi-core processors became common the MPI-only approach did not work very well because MPI ranks need to send and receive messages to transfer data, even if the ranks are housed in the same SMPs. For this reason, a master-slave model is adopted by Uintah runtime where MPI ranks have multi-threaded execution. The master thread does the data communication with other MPI ranks and other threads work on the computation. Later, the design of the scheduler was changed in Uintah to support a computation offload model where

Uintah can work on heterogeneous systems to offload work for CPUs and GPUs. However, the master-slave scheduler created problems such as the volume of work (communication and management) to be handled by the master thread. In 2013, Uintah launched a unified scheduler to address this problem, and as a result the scheduling was decentralized. The new scheduler can work asynchronously by creating DAG based execution model where a task can be scheduled in CPUs and GPUs simultaneously. The Uintah scheduler is capable of overlapping computation and communication through an asynchronous execution scheme of the DAG (the initial implementation was static). The heterogeneous master-slave scheduler maintains multiple queues for CPU and GPUs. The GPU queues are capable of maintaining multi-streams which enables the use of multiple GPUs. The data transfer from GPU to host is done by the runtime system. In the current scheduler, multiple CPU and GPU queues exist where tasks reside. The worker threads and GPUs pull tasks from that and executes.

G. PaRSEC

PaRSEC [26], [92], an effort from the University of Tennessee, Knoxville, was introduced in 2012. PaRSEC provides a dataflow programming model. The main idea is to express a program through dataflow between different parts of the code. When dataflow is defined, the dependencies get exposed. This representation of the dataflow acts as a hint to the runtime system for orchestrating the DAG execution on available hardware. At first, the sequential dataflow expressed by the application developer is translated using PaRSEC. The translated version is called JDF representation. The JDF representation is then pre-compiled into C code. The pre-compiled C is then linked with the application binary. For this reason, the source program and the JDF (the dataflow representation) become available to the runtime system. This particular feature is not common in other runtime systems. Since the dependencies are already part of the executable, PaRSEC can do necessary communication implicitly without any user intervention. The runtime implements non-blocking communication which facilitates computation and communication overlap.

VI. GAS Based Runtime Systems

In this section, GAS based runtimes are discussed. Table IV shows information about the GAS based runtimes.

A. UPC

UPC [96] is one of the pioneers of modern PGAS languages. It originated from LBNL in 1999. As previously mentioned, it is considered to be the descendent of SPLIT-C [38]. It provides an option for distributed data structures that are available for reading from and writing to different nodes. In other words, data structures reside in nodes but can be accessed from other nodes. UPC provides a fixed SPMD model where parallelism is fixed from the beginning of a program. UPC can be imagined as a collection thread executing in a globally shared address space. In the beginning, UPC was an extension of the C language. However, in 2014, UPC++ was released to support object-oriented design in C++, asynchronous execution with multidimensional arrays, and support for integrating other runtime systems to provide PGAS support [97]. Using the C++11 standard's async library, UPC++ provides the option for asynchronously accessing data structures through the global address space.

B. Chapel

Chapel [25] is a programming language that emerged from CRAY's effort in DARPA high performance computing system program (HPCS). Chapel [98] is a PGAS language (a separate language) which similar to high-level programming languages like C, Java, and Fortran that provides a global view of the system it is running on and supports a block-imperative programming style. The main reason for a new language, the authors argue, is to set the users in the right state of mind where users know that this is not a sequential program; instead, it is a parallel program. Chapel provides all the basics of high-level programming such as loops, conditions, types, etc. Along with regular high-level programming language constructs, Chapel provides parallel constructs such as domain (similar to array concept), parallel iteration forall loops. The domains (data structures) can be distributed among locales (nodes). Chapel supports task parallelism when specified by the programmer (using begin and co-begin statement). Data distribution is hidden, unlike UPC.

C. X10

The X10 language [47], [81] is a member of the PGAS family. It was introduced by the IBM Watson lab in 2005 as part of the DARPA High-Performance computing program (HPCS). X10 was introduced at the start of the many-core era, and was targeted for large shared multiprocessor (SMP) environments where processors would have non-uniform access to memory (NUMA). X10 introduces object-oriented facilities by having JAVA as the foundation for sequential programming languages. It was a risky decision because JAVA was not particularly popular in the HPC community. However, the assumption was, by 2010, JAVA developers would bridge the performance gap between Java and C/C++. X10's goal was to provide a way for programmers to go beyond standard JAVA constructs and provide HPC-specific constructs that do not depend on JAVA, such as asynchronous execution and multidimensional arrays. Moreover, X10 programs would run on top of the JAVA virtual machine, and at the lower level X10 would support shared memory models such as OpenMP and for communication and distributed memory. X10 would use MPI or RDMA communication for the asynchronous PGAS support. X10 provides distributed arrays through PGAS support, where different *places* (processors) can have local data and can access remote data. Through the X10 constructs *place* and *activity*, X10 can provide asynchronous tasking with an option for accessing globally available data structures. X10 also supports parallel loops.

Name	Who	When	Open source	Implementation	Target	One line description
UPC	UCB, LBNL	1999	Yes	Extension	Distributed	PGAS based parallel programming language for C
Chapel	Cray	2007	Yes	Language	Distributed	High level language for Global view and asynchronous tasks
X10	IBM	2005	Yes	Language	Distributed	A PGAS language on top of Java virtual machine

TABLE IV: Runtime systems for PGAS languages.

VII. Heterogeneous Runtime Systems

Heterogeneous runtime systems are discussed in this section. Table V shows the heterogeneous runtimes.

A. OpenCL

The Open Computing Language (OpenCL) [50] standard is managed by the Khronos group. The first specification for OpenCL 1.0 was released in 2009. OpenCL is designed for heterogeneous systems with different devices from different manufacturers. OpenCL provides queues for each device and the CPU is considered as host. The host can enqueue kernels for execution in a blocking and non-blocking way. The API provides means to transfer data between the host and the device and various synchronization functionalities. The abstraction layer provided by OpenCL makes creating scalable code for different vendors easy (all accelerators are devices). The OpenCL execution model has different hierarchies. When a device from a specific vendor is chosen, those hierarchical execution constructs are mapped to the underlying device driver.

In 2015, a C++ programming layer for OpenCL, SYCL was launched [99]. SYCL provides C++ single-source programming for both host code and kernels. Moreover, the Celerity [100] runtime system is developed using SYCL and MPI. The celerity runtime system can execute SYCL code in GPU clusters in a distributed memory environment. Celerity divides the data structures using specified mapping and executes them on different GPUs. The communication is taken care of by the runtime system and hidden from the user.

B. OpenACC

Realizing the popularity of the directive-based programming approach, Cray, NVIDIA and PGI developed the OpenACC [101] programming standard for accelerators in 2012. The main idea was to simplify parallel programming for heterogeneous CPU/GPU systems. High-level abstractions through directives hide all the detail of offloading a kernel to GPUs. Moreover, it ensures portability to different manufacturers. OpenACC treats every compute element as a device including CPUs. It reserves one CPU thread for host operation and unlike OpenCL, the rest of the threads can be used as a device where parallel execution can be performed in a shared memory environment. The concept of OpenACC is similar to OpenMP when it comes to serial and parallel regions. However, the host thread can asynchronously offload computation to the device and progress the execution. The host thread also can check the status of the queue for synchronization. Data transfer to and from a device is also done through directives. OpenACC is gaining popularity for its simplistic design.

C. StarPU

The StarPU [14] runtime system was introduced in 2011 by the Inria Institute, located in France. The main idea of StarPU is to provide a task-based programming model which is capable of heterogeneous execution (CPU/GPU). The main data structure of StarPU is called a *codelet*. A computational kernel is expressed as a *codelet* where the kernel can be executed in a CPU, CUDA device, or in an OpenCL device. A *task* is another data structure that is associated with a *codelet* and the dataset on which the *codelet* will execute. The dependency between tasks can be deduced by the runtime system from data dependency, or the dependency can be manually expressed. When a *codelet* is designed, a target device is specified. During execution, if a task is executed in CUDA or OpenCL device, the runtime system carries out the data transfer activity. Task execution in StarPU is nonblocking and hence, provides an asynchronous task execution model. Each device has a task queue and because of non-blocking communication, the queued tasks can be rearranged for better performance. A *codelet* with a CPU or GPU target device can be executed on either device, making StarPU a unified model for heterogeneous architecture. One of the strengths of StarPU is its multiple schedulers from which users can choose while launching the StarPU application.

D. CUDA

CUDA [49] is a platform and application programming interface developed by NVIDIA and was introduced in 2007. CUDA is the catalyst for bringing GPGPUs to high performance community. Because of its throughput-oriented approach, CUDA was capable of providing significant computation power. CUDA became popular quickly and now, CUDA devices are found in every large compute facility. CUDA devices have a large number of low-performance cores where CUDA threads run. CUDA implements the host and device concept where the host CPU is capable of offloading computation to a CUDA device through the CUDA API. CUDA provides different levels of synchronization.

E. HIP

Similar to CUDA, AMD launched its ROCm [57] platform for GPUs. The ROCm platform consists of different tools, compilers, and libraries. In 2016, AMD introduced the Heterogeneous Compute Interface for Portability (HIP) API for GPUs. The ROCm stack consists of user code at the top, the HIP API that expresses the programming model, the HCC compiler that compiles HIP code, the HSA API and runtime for AMD GPUs, and at the end, the amdkfd driver for AMD GPUs. In this document, the name HIP is used

Name	Who	When	Open Source	Implementation	Target	One line description
StarPU	Inria	2011	Yes	Library	Heterogeneous	A task based runtime for heterogeneous system
OpenCL	Khronos	2009	Yes	Library	Heterogeneous	Pioneer for open-source heterogeneous computing
OpenACC	Cray, NVIDIA, PGI	2012	Yes	Library	Heterogeneous	Directive based heterogeneous programming
CUDA	NVIDIA	2007	No	Library	NVIDIA GPU	NVIDIA's CUDA device driver
HIP	AMD	2016	Yes	Library	AMD and NVIDIA	AMD's API for AMD and NVIDIA GPUs

TABLE V: Heterogeneous runtime systems

to describe both the programming API and the driver. The HIP [16] C++ Runtime API and Kernel Language can create portable code that runs both on NVIDIA and AMD GPUs. HIP is considered lightweight and does not have much impact over coding directly in CUDA mode. HIP provides features such as templates, C++11 lambdas, classes, and namespaces. Moreover, The HIPIFY tool is capable of converting CUDA code to HIP code.

VIII. Runtime Feature Comparison

This section compares and contrasts different runtime systems based on their programming models, APIs, execution models, memory models, and synchronization strategies. Later in this section, runtime systems are compared based on their communication and distributed execution features.

A. *Programming API and model*

A programming API provides a means of expressing the programming model for a runtime system. Programming APIs that conform to a high-level language standard come in various forms. This layer is typically the highest level of abstraction provided by the underlying runtime system.

Programming APIs play a critical role in determining the usability of a runtime system. There is a trade-off between abstraction and control. On one hand, if the API provides a very high-level of abstraction, a user may unwillingly cede control of fine-grained optimizations to the runtime. On the other hand, if a user wants to control fine-grained optimizations through the API, the source code can lose readability. For example, in MPI programs, the memory mapping strategy and synchronization techniques are explicitly expressed in the code. Compared to modern runtime systems such as HPX, MPI does not provide a high level of abstraction. However, it offers the user full control over the key factors affecting performance. Modern runtime systems carefully balance this trade-off by providing different levels of control to empower users.

The APIs of modern HPC runtimes are written in a high-level language such as C/C++/Fortran/Java. Table VI shows the language and the compiler for different programming APIs. There are similarities in how these programming models are expressed. We describe these similarities below.

Runtime	Language	Compiler support
Cilk Plus	C/C++	Built in Intel compiler and others
TBB	C++	Built in Intel compiler and others
OpenMP	C/C++/Fortran	In all major compilers
Nanos++	C/C++	Mercurium for OmpSs
Qthread	C/C++	Standard compilers
Charm++	C/C++	Charm has it's compiler
HPX	C++	Standard C++11, 14, 17
Legion	C/C++/Regent	Standard/Regent compiler
OCR	C/C++/Fortran	Several implementation
Argobots	C/C++	OpenMP (GNU) and MPI
Uintah	C/C++	MPI+X
PaRSEC	C/C++	Own compiler for two stages
UPC	UPC	UPC compiler
Chapel	Chapel	Chapel compiler
X10	X10	X10-Java compiler
StarPU	C	Standard compilers
OpenCL	C/C++/Python	Standard compilers
OpenACC	C/C++/Fortran	Standard compilers
CUDA	C/C++/Fortran	nvcc compiler from CUDA
HIP	C/C++	hcc compiler from ROCm

TABLE VI: Programming model and API

1) Directive based: Directive-based programming models are favored by many runtimes for their capability to provide a high level of abstraction and the ease with which they allow the user to express loop-level parallelism. The two most common directive-based programming models are OpenMP and OpenACC. Both of these programming models provide execution schemes for the CPU and the GPU. To express parallelism, a user identifies a parallel region, and through a compiler directive (pragma), notifies the runtime of the execution strategy to implement. This simplicity has made OpenMP one of the most popular and ubiquitous programming models in high-performance parallel computing. The OmpSs programming model from Nanos++ and Bolt [90] from Argobots also provide a directive-based approach for expressing parallelism.

2) Expressing asynchronous execution: Many runtimes offer asynchronous execution. However, the programming model and the programming API provide the flexibility to the user to specify which portion of the code to run asynchronously. While some AMT runtimes such as Charm++, HPX, Cilk, and StarPU provide implicit asynchronous execution schemes based on data-dependency graphs and non-blocking execution, the APIs of other runtimes offer special constructs for asynchronous execution. Chapel introduces the “cobegin” construct that instructs the runtime system to execute the task in parallel. However, a descendent or child of the parallel task executes asynchronously depending on the implicit data-dependencies in the program. Similarly, OmpSs uses the “concurrent” construct to implement relaxed data-dependency. OCR uses event-driven tasks (EDT) for asynchronous execution. X10 uses “async” to create asynchronous tasks, while Charm++ and HPX use futures.

3) GPU programming: Programming models for enabling parallel execution on GPUs are significantly different from those that run on CPUs. OpenMP 4.0 and OpenACC successfully hide GPU-specific support, and the runtime system takes care of implementing those details. However, OpenCL, CUDA, and HIP provide a low-level programming API to express the GPU programming model. Table VII shows the similarities in how OpenCL, CUDA, and HIP allow the user to express parallelism. Generally, all of these programming models divide a GPU computation into a grid of thread-blocks. The runtime then maps these thread-blocks onto the streaming multiprocessors (SM) (AMD calls these "compute units" (CU)). While the concept is similar, the terminologies are different in OpenCL, CUDA, and HIP. OpenCL provides separate computation queues for the various GPU devices on the system. The OpenCL runtime also offers multiple computation queues for different heterogeneous compute elements. OpenCL supports both NVIDIA and AMD GPUs. ROCm provides a layer for translating CUDA code into HIP code that allows CUDA code to run on AMD GPUs.

Runtime	Grid	Thread Block	Thread	Warp
OpenCL	NDRange	work group	work item	sub-group
CUDA	grid	block	thread	warp
HIP	grid	block	work item/thread	wavefront

TABLE VII: Similarity between CUDA, HIP and OpenCL

4) New Language with special compiler: Some runtime systems offer compilers alongside their programming APIs. Chapel provides a compiler for its language. X10's compiler translates X10 code to Java or C++ code. Charm++ uses its compiler wrapper for Charm++ codes. PaRSEC also provides a pre-compiler to translate its data-flow representation of the task-graphs into C code. The programming APIs that are implemented as library and language extensions can be compiled through standard compilers. Some APIs use recent features of high-level programming languages. For example, HPX uses constructs from C++11, C++14, and C++17.

B. Execution model

The execution model refers to the actual execution scheme a program follows while executing. After the program is compiled, the binary has all the instructions for the runtime to shape its execution. Similar-looking code can behave differently depending on the underlying runtime system. All HPC runtime systems support distributed execution, usually through the SPMD execution model. However, the detail of the execution model varies. Table VIII provides an overview of the execution models supported by different HPC runtime systems. The commonly found characteristics are described below.

1) Task parallel vs Data parallel: In the task-parallel model, disctinct tasks execute in parallel. This is in contrast to the data-parallel model, which extracts parallelism from SIMD instructions. Historically OpenMP followed a data-parallel execution model before OpenMP 4.0, which introduced task parallelism. Task-parallel execution provides more flexibility for the user to extract parallelism from situations where data-parallelism does not apply. However, OpenMP tasks are heavy-weight OS threads which makes context switching slow. Thus, OpenMP provides tasking at a coarse-grained level. The same is true when MPI + X applications employ OpenMP. OpenMP follows a fork-join model. Even though OmpSs and Nanos++ are considered fore-runners of OpenMP, they do not support the fork-join model. However, Cilk, TBB, and Argobots follow the fork-join model.

Runtime	Execution model
Cilk Plus	Asynchronous task (DAG), Fork-join, SIMD
TBB	Asynchronous task (DAG), Fork-join, SIMD
OpenMP	Fork-join, SIMD
Nanos++	Asynchronous task (DAG), Fork-join, SIMD
Qthread	Asynchronous task (DAG), Fork-join, SIMD
Charm++	Message driven asynchronous execution. DAG of tasks
HPX	Message driven asynchronous execution. DAG of tasks
Legion	Asynchronous execution builds a Tree of tasks
OCR	Event driven Asynchronous execution. DAG of tasks
Argobots	Fork-Join execution that builds a DAG of tasks
Uintah	MPI + X DAG of Tasks
PaRSEC	Event driven Asynchronous execution.DAG of tasks
UPC	Pthreads with GAS (supports asynchronous execution)
Chapel	Asynchronous execution builds a DAG of tasks
X10	Asynchronous execution builds a DAG of tasks
StarPU	Asynchronous execution builds a DAG of tasks
OpenCL	Heterogeneous execution: different execution scheme
OpenACC	Heterogeneous execution: different execution scheme
CUDA	Data parallel execution
HIP	Data parallel execution

TABLE VIII: Execution model

2) Asynchronous many task parallel: Because OS threads are bulky and creating and destroying them incurs a prohibitively high overhead, using them within many task runtimes is not feasible. For this reason, many task runtimes use lightweight tasks. These tasks can be a simple function call or a group of instructions within a function. These tasks are easy to create and destroy, and they also yield quickly. Asynchronous execution runtimes which employ lightweight tasks take advantage of the fact that these tasks leave a small memory footprint to reduce context-switching latency. The asynchronous many task model of execution has become popular in modern runtime systems. Such execution models create a graph of tasks with and without dependency among the nodes. In such a model, the number of tasks can be in the range of millions (some distributed memory runtimes report handling up to 100 million tasks). Cilk, TBB, and Qthreads are examples of such many task runtimes. As they employ a shared memory model, the number of tasks is significantly smaller when compared to the distributed memory model. Runtime systems that can work with distributed memory architectures such as HPX, Charm++, and Legion can spawn a very high number of tasks to provide parallelism. OCR, Argobots, PaRSEC, Chapel, X10, and StarPU also follow the many task execution model.

3) Message driven vs message passing: The message-passing model expresses code in an SPMD model. Messages allow the transfer of data between two processes. In such a model, computation is in the driving seat. For example, the MPI + X model follows the message-passing paradigm. Runtime systems such as Uintah and Argobots follow the message-passing model. In the message-driven model, the data dependency dominates program execution, and messages coordinate the execution flow. HPX and Charm++ follow the message-driven execution model. Data or a message is sent to different *chares* in Charm++, while HPX sends computation towards the data. Moreover, there is another model called event-driven execution that is similar to the message-driven paradigm. OCR and PaRSEC belong to this category.

4) GPU execution: The GPU execution model follows a single-instruction, multiple-thread (SIMT) model. As shown in Table VII, the runtime scheduler assigns thread blocks to different streaming processors. The runtime then divides the threads into a set of warps (32 threads form a warp on an NVIDIA GPU and 64 threads in AMD GPUs) [102]. These warps or wavefronts execute in SIMD fashion. OpenACC and OpenCL provide a high-level construct to utilize the heterogeneous platform. Both OpenCL and OpenACC employ device-wise queues to enqueue different kernel executions.

C. Memory model and synchronization

Memory handling is one of the main bottlenecks of achieving high performance since synchronization is needed when multiple compute entities try to access the same memory. Moreover, synchronization is necessary for both the application developer to express the computation in a well-coordinated manner, and also for the runtime system to coordinate with different computing entities. Early runtime systems provided synchronization constructs where the entire program synchronized both in distributed memory and shared memory systems. This is referred to as the bulk synchronous model. Bulk synchronous models are easy to implement and understand. However, they suffer from performance penalties and also reduces the utilization of the system. For this reason, asynchronous models are now popular in the runtime community, as they provide fine-grained synchronization to improve utilization. Table IX shows the memory model and synchronization in runtime. The memory model is discussed first, followed by a discussion of synchronization in the runtime systems.

1) Memory model: In the shared memory model, all the processors share the same cache-coherent memory space. Every compute element can access any memory location through the memory channel without communicating through the network interface. Cilk, TBB, OpenMP, OmpSs, and Qthreads are examples of runtime systems that run on shared memory. Threads can access shared data structures where they can read and write. However, individual threads have their own memory space and private data. In a distributed memory model, network interfaces connect different nodes. OCR, Argobots, Uintah, and PaRSEC have distributed memory runtimes. In distributed memory runtimes, over-the-network communication is necessary to access remote memory. However, this communication is not explicitly visible to the user, and the runtime system performs the communication underneath the hood.

Runtime	Memory model	Synchronisation
Cilk Plus	Shared memory	Cilk join
TBB	Shared memory	Mutex
OpenMP	Shared memory	Directives
Nanos++	Shared memory	Directives
Qthreads	Shared memory	Mutex and FEB
Charm++	Distributed shared memory	Message, Futures
HPX	Distributed shared memory	LCOs
Legion	Distributed shared memory	Custom locks
OCR	Distributed memory	Events, Data-Block
Argobots	Distributed memory	Mutex, futures
Uintah	Distributed memory	Mutex, MPI
PaRSEC	Distributed memory	Dependency based
UPC	Distributed shared memory	Locks, barriers
Chapel	Distributed shared memory	Sync, single
X10	Distributed shared memory	Clocks
StarPU	Heterogeneous memory	Locks, barriers
OpenCL	Heterogeneous memory	Barriers
OpenACC	Heterogeneous memory	Directives
CUDA	GPU memory	Library call
HIP	GPU memory	Library call

TABLE IX: Memory model and synchronisation

Charm++, HPX, Legion, UPC, Chapel, and X10 provide a distributed shared memory space programming model. Within a compute node consisting of GPUs, the PCIe bus connects the GPU and CPU memories. Data transfers between the CPU and GPU memories are required to share memory between the two devices. StarPU, OpenACC, and OpenCL operate through this type of GPU-CPU shared memory model. CUDA and HIP operate within a GPU memory structure that hosts a global memory, a local memory, and a shared memory. Both global memory and local memory are slow. Global memory is accessible by all the threads, whereas local memory is local to every thread. Shared memory resides on a streaming multiprocessor (SM), and the threads of the thread block share this memory.

2) Synchronisation: As described in the previous section on execution models, many runtime systems support asynchronous execution where the sequence of execution is non-deterministic. In such execution scenarios, synchronization happens at the task level based on the DAG of dependencies. On the one hand, Cilk, TBB, OmpSs, and Qthreads provide fine-grained synchronization since asynchronous execution is allowed in these runtimes. Cilk provides join [53], TBB provides atomic locks and mutexes [103], OmpSs provides data dependencies and directives [70] and Qthreads provides mutex and FEB for synchronizing the tasks [76]. On the other hand, OpenMP synchronizations are not fine-grained. OpenMP provides directive-based barriers [22]. HPX also uses local control objects such as futures, dataflow objects, etc., that implement the synchronization in a way that ensures that the tasks can keep working without being completely blocked [23]. Charm++ uses messages for data synchronisation [79] and uses futures for task synchronization.

For data access, Legion provides an option for relaxed synchronization. There are two types of coherence: exclusive coherence and relaxed coherence. In exclusive coherence, the synchronization is strict, where Legion follows an order.

Runtime	Communication	Distributed support	GPU support
Cilk Plus	None	No (through MPI+X)	NA
TBB	None	No (through MPI+X)	NA
OpenMP	None	No (through MPI+X)	2013
Nanos++	None	No (through MPI+X)	2011
Qthread	None	No (through MPI+X)	NA
Charm++	RDMA	Yes (through GAS)	2016
HPX	Parcel	Yes (through GAS)	2014
Legion	GASNet	Yes (through GAS)	2012
OCR	MPI messages	Yes (through MPI)	NA
Argobots	MPI Messages	yes (MPI + X)	2016
Uintah	MPI Messages	yes (MPI + X)	2013
PaRSEC	MPI messages	yes (Through MPI)	2012
UPC	GASNet	Yes (through GAS)	2014
Chapel	GASNet	Yes (through GAS)	2019
X10	MPI Messages	Yes (through MPI)	NA
StarPU	None	No (through MPI+X)	2011
OpenCL	None	No (through MPI+X)	2011
OpenACC	None	No (through MPI+X)	2012
CUDA	None	No (through MPI+X)	2009
HIP	None	No (through MPI+X)	2016

TABLE X: Communication, distributed support, and GPU support

However, in relaxed coherence, the synchronization order is not maintained. Rather, Legion ensures access. Atomic coherence serializes the access without ordering, and as the name suggests, simultaneous coherence lets two threads partially execute simultaneously. Legion provides reservation (small scope) and phase barriers (user-defined larger scope) for synchronization [82]. In OCR, events are the main synchronization point since OCR uses an event-driven paradigm. For data synchronization between tasks, OCR uses data blocks where access priority determines the serialization of data accesses [58]. PaRSEC provides dependency-based synchronisation [104]. UPC uses locks and barriers for synchronisations [105]. Chapel uses full or empty syntax for synchronization. It supports two types of synchronization variables: *sync* and *single* [25]. The *sync* variable switches state from full to empty for access control, and the *single* variable indicates that it can only be read once. X10 uses clocks for synchronisation [47]. Clocks ensure deadlock-free operation by waiting for some time and then releasing. StarPU provides locks, nested locks, critical sections, and barriers for synchronization [52]. OpenCL provides three types of barriers. The first kind is to ensure that OpenCL executes all the items in the queue. The second kind synchronizes all the work-items in a work-group on a device. The third kind ensures the synchronization among sub-groups [50]. OpenACC provides directive-based synchronisation through barriers [101]. CUDA and HIP provide synchronization for devices, streams, and threads [16], [49].

D. Communication, distributed support, and GPU support

Communication is necessary to provide distributed support. Table X shows communication mechanisms and distributed execution options for runtime systems. It also shows the time when a runtime system first reported support for GPUs.

Efficient communication is a prerequisite for ensuring high performance in runtimes. Runtime systems either use their own communication framework or use already existing, optimized libraries. Charm++ uses messages for communication. It supports different communication libraries such as MPI and UDP. Charm++ also provides the option for one-sided communication in an RDMA-enabled network. Through the communication interface, Charm++ provides a global address space for supporting distributed execution [89]. HPX consists of a Parcel subsystem that carries out communication across nodes. The Parcel subsystem can communicate using TCP ports or MPI. By using active messages through the Parcel subsystem, HPX enables a global address space for distributed execution [23]. Legion, UPC, and Chapel use the GASNet [46] one-sided communication library for distributed execution. PaRSEC and X10 use MPI messages for communication. The shared memory and accelerator runtimes do not use communication across nodes. However, they can be a part of the MPI + X execution model to support distributed execution.

Most of the modern runtimes support GPUs. Runtime systems use a CUDA, HIP, or OpenCL runtime to provide GPU support in their ecosystem. Cilk Plus, TBB, Qthread, OCR, and X10 do not report support for GPUs.

E. Discussion

Two trends are visible in the runtime system domain. The first trend is the adoption of an asynchronous task-based approach for improving overall resource utilization. The second trend is including heterogeneous capabilities. The first trend shows the direction where the runtime system community is heading, whereas the second trend results from hardware improvements. For this reason, the HPC community has more control over the first trend. However, the task-based runtimes have not yet agreed upon a standard. Every runtime system has its methodologies for implementing the asynchronous task-based approach. Initiatives such as OCR showcase attempts from the HPC community to design a specification for task-based systems. Argobots is another research effort to bring a variety of runtime systems under one umbrella. This trend suggests the need for standardization of asynchronous many task execution approaches.

IX. Dynamic adaptation in Runtimes

This section explores the dynamic features of modern HPC runtime systems. Specifically, we consider the features that are primarily responsible for providing better performance and energy consumption.

A. Scheduling and load balancing

Scheduling is one of the most critical tasks that an HPC runtime performs. In early parallel programming models, scheduling used to be mostly static. After expressing the parallelism through programming APIs in early MPI or OpenMP applications, the mapping between work and resource did not change during execution. However, in modern HPC runtimes, the scheduling scenario is highly dynamic. It is almost impossible to determine the optimal mapping between work and

Runtime	Scheduling	Load balancing
Cilk Plus	Tasks on Worker thread pool	Yes (work-stealing)
TBB	Tasks on Worker thread pool	Yes (work-stealing)
OpenMP	Heavy OS threads	Yes (work-stealing)
Nanos++	Tasks on Worker thread pool	Yes (work-stealing)
Qthread	Tasks on Worker thread pool	Yes (work-stealing)
Charm++	Tasks on Worker thread pool	Yes (through migration)
HPX	Tasks on Worker thread pool	Yes (through migration)
Legion	Tasks on Worker thread pool	Yes (work-stealing)
OCR	Tasks on Worker thread pool	Yes (work-stealing)
Argobots	Stacked custom scheduling	Yes (work-stealing)
Uintah	Tasks on Worker thread pool	Yes (Dynamic adaptive)
PaRSEC	Tasks on Worker thread pool	Yes (work-stealing)
UPC	None	None
Chapel	Future plan	Future plan
X10	Tasks on Worker thread pool	Yes (work-stealing)
StarPU	Multiple	Yes (work-stealing)
OpenCL	Heterogeneous queues	Dependent
OpenACC	Heterogeneous queues	Dependent
CUDA	GPU scheduling	Yes
HIP	GPU scheduling	Yes

TABLE XI: Scheduling and load balancing

resources since the optimal fine-grained mapping keeps changing. The main idea behind this non-deterministic mapping is to increase the utilization of underlying hardware. As a result, scheduling is one of the areas where dynamic adaptation plays an important role. Dynamic scheduling enables better load-balancing that, in turn, results in better system utilization. We discuss the scheduling and load balancing strategies of different runtime systems below.

Table XI shows the scheduling approaches adopted by different runtime systems. Each category presented in the table is further elaborated in the following sections.

1) Scheduling using OS threads: OpenMP uses direct task mapping on OS threads. Every time OpenMP creates a task, OpenMP assigns the task to an OS thread. This OS thread is created at the beginning of the parallel region and joins with the master thread when the parallel region ends. However, OpenMP is unaware of the task-to-thread mapping strategy to implement in advance (it can be specified). OpenMP standard provides five types of scheduling: 1) static, 2) dynamic, 3) guided, 4) auto, and 5) runtime [22]. It also provides the option to change the chunk size. The number of loop iterations each OS thread gets assigned depends on the chunk size. Static scheduling distributes the number of iterations equally if the chunk size is not specified. If the chunk size is specified, OpenMP allocates chunks to different threads in a round-robin fashion. In dynamic scheduling, each thread works on an initial chunk and requests more chunks as required. Guided scheduling works like dynamic scheduling, except that the chunk size keeps decreasing. When the user specifies auto as the scheduling option, the compiler decides the data distribution. When the runtime scheduling is selected, OpenMP determines the chunk sizes at runtime. Using dynamic scheduling, OpenMP can achieve work-stealing load-balancing. OmpSs [70] also implements the same strategies.

2) Worker thread pool: The most common strategy for scheduling in many task runtime systems is to have a pool of worker threads. The runtime system has task queues that contain ready-to-be-executed tasks. Similar to the producer-consumer approach, when a task's dependencies are resolved, it is placed on the ready queue. The pool of worker threads keeps pulling tasks from the ready queue. A majority of the many task runtimes implement this strategy. Cilk Plus, TBB, Nanos++, Qthread, Charm++, HPX, Legion, OCR, Uintah, PaRSEC, and X10 all implement some variant of this strategy. The main benefit is the increased utilization of the resources. However, the queue structure and the number of queues differ in different runtime systems. Uintah implements a unified schedular where MPI, Pthread, and CUDA can work together in an out-of-order fashion where the pthreads are the worker pool that consumes work from the CPU queues. It has a scheduling option for MPI processes as well. The load balancer in Uintah can provide dynamic adaptation in runtime by changing how much computation each processor performs [95]. Nanos++ holds a ready task queue where all the tasks have their dependencies resolved (supports yielding) [70]. Qthread employs a similar strategy where the worker pool is called a "collection of a shepherd" (uses chunk size) [75].

In Charm++, each PE (worker thread) has its pool of messages and a collection of chares. As Charm++ employs a message-driven paradigm, each PE selects a message from the pool and executes the method of a chare for which the message is meant for. Charm++ provides an advanced load balancing strategy through migration. It can provide load balancing in a centralized or in a distributed way. It also employs a measurement-based load balancing strategy. Charm++ creates a database of information that facilitates periodical load balancing using prediction of the imbalance. It also provides different algorithms for load balancing (Greedy, Refine, Rotate, etc.) [89]. Like Charm++, HPX also keeps queue(s) of tasks for each OS thread. HPX also provides multiple priority queues where HPX executes high priority queues first. HPX provides different scheduling options (Priority local scheduling, Priority ABP scheduling, etc.). Through the Priority ABP scheduling, HPX can provide NUMA sensitive scheduling where HPX assigns the highest priority to the same NUMA domain is given high priority [23]. HPX provides a load balancing option for in-node through work-stealing among the worker threads and also distributed load balancing through task migration [106]. In Legion, the underlying Realm runtime manages the worker thread pool. This pool creates a queue for each thread and asynchronously executes them [84]. The mapping interface of legion runtime provides a mechanism for distributed task-stealing for load balancing [107]. OCR also uses a worker thread concept where the load balancing is supported through work-stealing using a work-first or help-first mechanism. The Habanero runtime (an upgrade of X10) [108], [109] is the inspiration for OCR's load balancing strategy. Argobots uses a stacked scheduler concept where multiple schedulers can be applied for different software modules during execution [59]. While using a set of worker threads, it also uses work-stealing load balancing. Like HPX, PaRSEC and Nanos++ also provide NUMA-aware scheduling for better performance [70], [92]. PaRSEC also supports inter-node and intra-node load balancing using work-stealing [92]. X10 provides load balancing custom work-stealing method through GLB library [110].

3) GPU scheduling: Both CUDA and HIP provide streams for devices. Streams execute the kernels sequentially in a first-come, first-served manner [111]. However, at a lower level, each streaming processor (SM) schedules warps (a set of threads) from the assigned thread blocks [102]. Each streaming processor has multiple warp schedulers that pull ready warps to execute from the queue to increase utilization. When a kernel starts executing on the GPUs, the scheduler assigns thread blocks to SMs. Much detail of the scheduling at the SM level is not revealed [111].

4) Heterogeneous scheduling: Scheduling in StarPU also follows a group of workers where the workers can be accelerators as well [14]. The default schedular in StarPU is a work-stealing scheduler. However, StarPU has different options for scheduling. StarPU can implement performance models to find out the appropriate target for specific tasks. Moreover, when declaring "codelets" in StarPU, the user can specify a priority for tasks that acts as a hint to the runtime. Based on these hints, StarPU schedulers can provide greedy scheduling [52]. The gang, worker, and vector constructs define the scheduling in OpenACC [112]. Based on the specified size of these variables underlying heterogeneous processors is used. Further, the underlying driver for the device implements the scheduling decision. OpenCL provides device-wise queues for heterogeneous systems [113]. When scheduled to the device queues, the device uses its internal scheduling at the execution time.

5) Discussion: The StarPU runtime system provides some scheduling mechanism for task placements on CPU and GPUs. Other runtime systems do not do so. However, the capability to operate on heterogeneous systems has become common in almost all modern asynchronous task-based runtimes. For this reason, a heterogeneous task placement scheduler would be a fruitful addition to task-based runtimes.

B. Energy aware features and studies

Energy consumption is one of the biggest concerns surrounding the operation of exascale systems [18]. For this reason, new processors (both CPU and GPU) come with predefined TDP levels and frequency sets. These new technologies enable the processors to dynamically adjust their clock frequencies to ensure that they adhere to the power budget. Moreover, CPU and GPU vendors provide interfaces to monitor and allow changing these states through those interfaces. For example, Intel provides running average power limit (RAPL) [114] and NVIDIA provides NVIDIA management library (NVML) [115] to monitor and control power-related attributes. These interfaces enable runtime systems (or OS) to select certain settings to limit energy consumption by sacrificing processing power. Having this "soft" control enables the runtime system to dynamically select the energy consumption mode depending on the priority, need, or hardware status. Such control has proved beneficial as it provides an extra layer of control to make energy-aware decisions. This section discusses energy-aware capabilities in runtimes and methods. At first, we discuss the most common energy-aware techniques suitable for runtime systems. Later, we present a discussion of the energy-aware decision capability that exists in current runtimes.

1) Dynamic voltage and frequency scaling (DVFS): Dynamic voltage and frequency scaling (DVFS) is one of the oldest methods to achieve dynamic power behavior. Many of the current processors have DVFS capability. In a DVFS capable system, processors and memory have a set of frequencies in which they can operate. In most of the devices, the frequency is selected by the operating system when DVFS is enabled. Usually, the frequency selection depends on the utilization of the unit. There has been a considerable amount of research done in the area of DVFS. Ma et al. [116] designed a GreenGPU that dynamically throttles the frequency of the GPU and the memory. Komoda et al. [117] also studied power capping using DVFS to find near-optimal frequency settings for CPU-GPU. Liu et al. [118] designed an energy-aware kernel mapping strategy that assigns different frequencies to PUs in a heterogeneous system using DVFS.

2) Power capping: Power capping is a technique that restricts the instant power consumption of a device. The main components of a system are the processors and the memory. Each processor has a certain number of frequencies that it can operate in, and the same is true for the system memory. Selecting a higher frequency guarantees a higher speed for the processor or memory but also consumes more power. For this reason, by opting for a lower frequency, the runtime can limit the maximum power consumption of a device instantly. A modern integrated device such as the NVIDIA Xavier has a pre-defined power cap. For example, Xavier has five pre-defined power caps that the runtime software can dynamically invoke. For example, Zhu et al. [6] dynamically finds the appropriate frequency to keep the application execution under a power cap for a heterogeneous system consisting of a CPU and a GPU. Using a machine learning technique, the strategy proposed by the authors can select a frequency that is capable of keeping a device under a power cap.

3) Energy-aware features in runtimes: Some runtimes include energy-aware features in their design. Charm++ provides the capability to change the CPU core frequency using DVFS [79]. Charm++ provides a load balancer that monitors the average temperature of the chip and changes the core frequency when the temperature crosses a threshold. These thresholds are application-specific, and the user can set them. When the frequency is lowered for a set of cores, the runtime calculates the load of the processor cores and identifies under-utilized and overloaded cores. After identification, the runtime load balancer migrates tasks from the overloaded to under-loaded cores. The runtime repeats this process many times during the application's execution. This approach provided energy savings without much performance penalty [119]. HPX does not provide energy-aware features, but they claim to

Runtime	Interface and online tools	Adaptation capability
Cilk Plus	No	No
TBB	No	No
OpenMP	OMPT interface	Exists
Nanos++	Event collection	No
Qthread	RCRdaemon tool	Exists
Charm++	PICs tool	Exists
HPX	APEX tool	Exists
Legion	Profiling Interface	No
OCR	No	No
Argobots	No	No
Uintah	No	No
PaRSEC	No	No
UPC	GASP interface	No
Chapel	No	No
X10	No	No
StarPU	Profiling Interface	No
OpenCL	Profiling Interface	No
OpenACC	Profiling Interface	No
CUDA	CUPTI (profiling)	No
HIP	roc-profiler library	No

TABLE XII: Tools and interface for dynamic adaptation

improve overall energy efficiency by increasing the utilization of the resources through over decomposition of tasks [12]. The same argument is made by OCR [86]. PaRSEC provides integration with PAPI [120] for power measurement at the task level. However, the runtime does not dynamically adapt itself using the power measurements [121]. StarPU provides energy-aware scheduling where the runtime system turns off the CPU cores to save energy. A "codelet" can be specified with an energy model, and based on that model, the runtime system adjusts the task distribution [52]. Other runtime systems do not provide energy-aware capabilities.

4) Discussion: Based on the runtime capabilities described here, runtime systems can provide efficient energy-aware decisions and offer a good trade-off between energy and performance. At the same time, it is also evident that not many runtime systems provide energy-aware features. For this reason, an energy-aware runtime that works well with hardware from different manufacturers would be a critical feature for future exascale systems.

C. Dynamic adaptation tools and interface

The runtime system is an active component that can interact with external entities. Such interaction can impose control on the decisions taken by the runtime. However, proper APIs need to be exposed by the runtimes for external systems to interact and influence their behavior. Table XII provides a summary of the different tools and interfaces for dynamic adaptation of the runtime systems. We discuss these in detail below.

1) Interfaces for runtimes: Some runtime systems reveal interfaces for the sake of collecting performance data during execution. Through these interfaces, runtimes allow an external entity to register callback functions to provide the status or value of different runtime variables. Having such a generic interface defined enables tuning runtime variables during execution. MPI 3.0 specification included MPI_T interface that allowed the community to design runtime introspection tools to change different parameters for efficient communication [122], [123]. Similarly, OpenMP 5.0 included OMPT, which is a tool interface for OpenMP. Similar to MPI_T, OMPT provides the ability to register callbacks to get the status of various runtime system parameters and timers [18], [19]. UPC provides the GASP [124] interface for registering callbacks. However, GASP does not provide the flexibility to change any runtime variables. Rather, GASP provides the facility for other tools to collect data (similar to the PMPI profiling interface in MPI). CUDA also provides the CUPTI profiling interface, but it does not provide an option for changing runtime variables [125]. The roc-profiler [126] of ROCm collects GPU performance data from AMD's HSA runtime [127]. StarPU also provides an online profiling interface and does not provide the option to change runtime variables [52]. Both OpenACC and OpenCL also provide a profiling interface designed only to enable querying and collecting runtime events [50], [101]. Similarly, Legion provides a profiling interface where the status of memory and tasks, execution time, and current load of the system can be obtained [107]. The Nanos++ and OmpSs ecosystems also provide an instrumentation option that can provide runtime events [70].

2) Dynamic adaptation tools for runtimes: APEX [128] is an autonomic performance measurement and analysis tool designed for task-based runtimes. It has support for HPX and OpenMP runtimes. APEX hosts a policy engine that can monitor runtime events and activate a policy based on that. Moreover, APEX can also implement a periodic policy. The APEX policy engine uses the Active harmony library [129] to change runtime parameters and observe their impact on performance. If APEX finds that performance improves, the policy engine continues modifying the runtime knob until it finds a near-optimal solution. Charm++ provides PICS [130] which can optimize application performance based on a control-point centric mechanism. Similar to APEX, PICS also collects information from the runtime system about the overall status. Unlike APEX, PICS employs control points both in the application and the runtime. Using a decision tree, PICS can tune different applications and runtime knobs based on the observed data. The RCRdaemon [131] can work with the Qthreads runtime. RCRdaemon sits continuously monitors the memory and utilization status from the OS. When the Qthreads scheduler starts execution with its worker thread pools (Pthreads), the adaptive scheduler in Qthreads communicates with RCRdaemon to find the optimal number of threads.

3) Discussion: While most runtime systems provide profiling interfaces, only a few expose APIs or provide a tool for enabling dynamic adaptation. Even though many runtime systems are open-sourced, building a custom tool for dynamic adaptation ties that tool to that particular runtime. For this reason, modern many task runtimes need a general interface specification solution such as MPI_T or OMPT.

X. Dynamic Adaptation Opportunities

The evolution of the runtime system suggests, major changes happened due to the memory model of parallel computing architectures. On the other hand, the recent trend

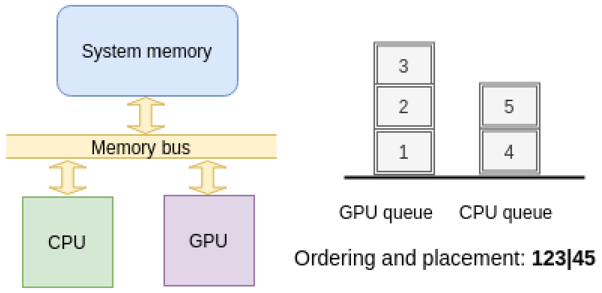


Fig. 6: A logical representation of iSMHS with CPU and GPU and Kernel queues for ordering and placement: *123|45*

suggests the continuation of a heterogeneous environment in the future. For this reason, research opportunities that combine these two need to be explored.

A. Open problem - memory contention mitigation in integrated heterogeneous systems

Integrated shared memory heterogeneous architectures are widely employed to satisfy the diverse needs of computing. While specialized processing units (PU) sharing one system memory improve performance and energy-efficiency by reducing data movement, they increase contention for memory as the PUs interact with each other. Collocated kernel execution on an iSMHS is portrayed in Fig. 6. In this system, the CPU and GPU are connected to a shared memory and the GPU does not have a private memory. For such a system, a case is shown where 5 ready-to-execute kernels with the ordering and placement (O&P) configuration of *123|45*, where kernels 1, 2, and 3 will be executed in order on the GPU and 4 and 5 will be placed for CPU execution. Kernels placed in different queues have the potential to execute in a collocated manner and may result in contention on the memory bus. For this reason, heterogeneous scheduling is impacted by memory contention. Several studies are presented here which are not yet part of HPC runtimes but bear the potential to be included.

1) Using machine learning: Machine learning is a powerful tool and is used to predict contention by some researchers. In [132], Zhu et al. studied co-scheduling on an integrated CPU-GPU system. The authors used a staged interpolation method to predict the execution time degradation. To categorize different kernels, the author used bandwidth as a metric.

2) Using algorithms: Algorithmic solutions are also explored by researchers. Finding the combination of kernels that minimizes memory contention requires searching for every combination. This problem is NP-Hard. For this reason, some researchers considered greedy solutions and argued that they can lead to a reasonable result. For example, Zhu et al. [6] devised a greedy algorithm that addressed memory contention from degradation in the execution time perspective while selecting a frequency for power capping. However, they did not consider the impact of memory contention on power or energy. They showed their greedy approach with post-local refinement can lead to a reasonable solution.

3) Addressing bus contention: As mentioned earlier, in some integrated heterogeneous systems, all processors share the same bus to access system memory. For this reason, contention occurs at the system bus level. Some researchers worked on the bus contention. Cavicchioli et al. [133] studied different SoCs and fused CPU-GPU devices to characterize memory contention. The author studied contention at a different point in the SoC systems, including bus contention. The study done by this author was execution-time oriented.

4) Addressing LLC contention: First-generation integrated heterogeneous systems previously had a shared last-level cache between processors. Having a shared cache among the CPU and GPU is adopted from the fact that multiple cores in CPUs share the cache hierarchy. While this makes the design simpler, it introduces contention in the Last Level Cache (LLC) when multiple processors heavily use the cache. Damschen et al. [134] studied memory contention and stalling in heterogeneous systems with shared the last-level cache (LLC). They showed because of the contention collocating multiple kernels in CPU and GPU at the same time does not yield good performance. Rai et al. [135] and Pan et al. [136] designed LLC management strategy for better performance. Memory contention due to shared LLC has also been studied by Garcia et al. [137] and Mekkat et al. [138].

5) Addressing contention using performance models: Some researchers used performance models to predict memory contention, while other researchers relied on machine learning to form empirical models. Examples of the empirical model include the work reported in [139]. There has been some effort on analytical modeling to predict memory contention. Hill et al. [140] extended the Roofline model for mobile SoCs to address memory contention from the perspective of bandwidth usage by the PUs.

6) Distribution of tasks based on the irregularity in workloads: Heterogeneous manycore and multicore systems expose the opportunity to seamlessly distribute workloads. This facilitates the re-shaping and re-distributing of a workload based on the type of workload, and provides the opportunity to select the appropriate processor. For example, irregular workloads are known to be more suited for CPUs and perform worse in a GPU because of the workload's memory access pattern. Cho et al. [141] addressed this problem and devised an on-the-fly strategy to partition irregular workloads in integrated CPU-GPU systems. At runtime, their strategy is capable of detecting irregular parts of the applications and scheduling them in the GPU, while the regular part of the load is scheduled on the CPU. They implemented their strategy in the OpenCL runtime. Zhang et al. [142] designed an irregularity-aware fine-grained workload partitioning technique. This work also finds the irregular execution chunks of a given application and runs the chunks in CPUs. The rest of the computation is done in GPU which provides better results. Pandit et al. [143] also looked at a similar problem and designed a dynamic work distribution considering the data transfer need of kernels in OpenCL runtime.

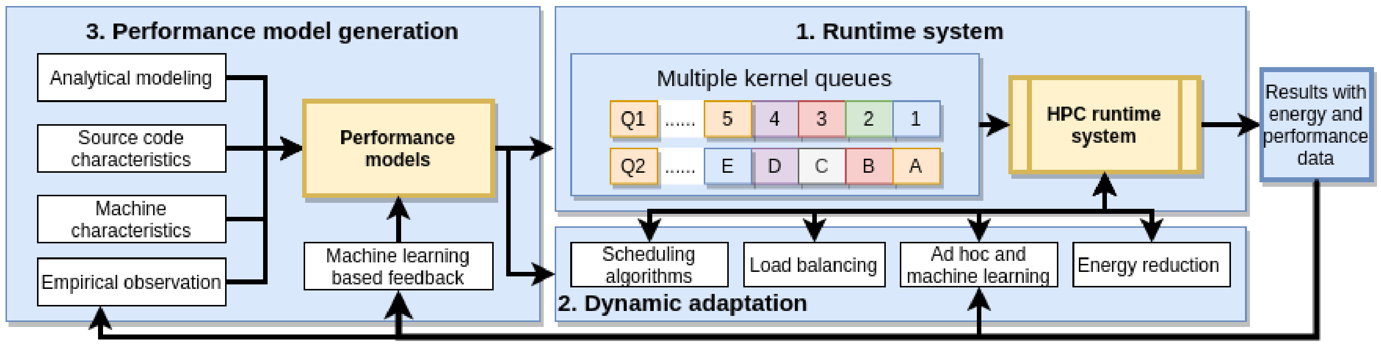


Fig. 7: Possible flow of performance improvement in an HPC runtime systems by using a performance model.

7) Scheduling based on profiling: Kaleem et al. [144] studied scheduling using profiling for load balancing between the CPU and GPU. They used a different version of profiling to improve on the current method and to fetch the kernel characteristics efficiently.

8) Individual workload placement on a processor: In a pipelined execution scheme, it is often important for a runtime to find the appropriate processor for the next workload. To find a suitable processor, both the application characteristics and current system status need to be considered. Panneerselvam et al. [3] devised a task placement strategy in a CPU-GPU system, where the runtime system monitors utilization of the system and decides a suitable placement that meets the applications' performance goals. The author also designed a model that predicts the standalone execution time of an application in a processor, which is then combined with the system utilization to make a decision.

Furthermore, Zhang et al. [145] extensively studied 46 applications to find out which application provides better results when collocated. Based on a decision tree-based model, their approach is capable of finding out kernel collocation impact for different applications in integrated CPU-GPU systems. The authors further showed the accuracy of the model to decide placement for an application in CPU or GPU in an integrated system.

9) Research opportunities: These works indicate the research community's involvement in scheduling decisions between CPUs and GPUs in new heterogeneous ecosystems. Some of them are tested with heterogeneous runtime systems like OpenCL [141]. Moreover, heterogeneous placement has energy consumption implications as well [146]. For this reason, energy- and performance-aware task placement in CPUs and GPUs should be considered in modern runtime systems.

B. Open problem - memory-centric performance modeling

In order to make an efficient decision of task placement in a suitable processing element (CPU or GPU), the runtime system should be equipped with performance models for generating kernel-level prediction. The main difficulty in making a performance model for predicting execution time or energy is the complex memory hierarchy in modern HPC compute resources. This is because memory traffic is the slowest component and heavily impacts both energy and performance. For this reason, memory-centric performance modeling is necessary. Some studies for memory access prediction for CPUs and GPUs are discussed below.

1) Memory access prediction: Several studies investigated memory access patterns to make a reasonable prediction. Yu et al. [147] used an analytical model of different memory access patterns for investigating application vulnerability. In Tuyere [148], Peng et al. used data-centric abstractions in an analytical model to predict memory traffic for different memory technologies. Moreover, categorizing and understanding memory access patterns plays an important role in Roofline models [149] where kernels are defined based on the ratio of flops and memory access between LLC and DRAM. Allen et al. [150] investigated the impact of different memory access patterns on GPUs. Some previous works used load and store instruction counts to measure memory access and used that count to predict performance(e.g., COMPASS by Lee at al. [151]). Compile-time static analysis, such as Cetus [152] and OpenARC [153], is also used to record instruction-level counts, which can then be used to measure performance. However, instruction-level counts do not reflect the role of the cache hierarchy. The above studies indicate that memory access prediction is studied but still an open problem in the HPC domain and should be studied further to empower runtime system decisions.

C. Discussion

Inspired by the open problems presented in this section, an example of dynamic adaptation in HPC runtimes can be viewed in Fig. 7. Box 1 represents the runtime systems with different kernel queues. Box 2 represents the dynamic adaptation engine which interacts with the runtime to facilitate scheduling, load balancing, and energy consumption reduction. Box 3 represents the offline performance model generation to feed the kernels of the runtime system and the dynamic adaption engine.

XI. Conclusion

Since the beginning of parallel computing, HPC runtimes have gone through major changes. These changes are caused by new architectures, increasing compute capabilities, upgrades in interconnect technologies, and the introduction of heterogeneity. Moreover, continuous innovation by the community and the decision to come together to standardize popular programming models had a major impact in shaping today's runtimes. Increasing layers of abstraction are observed which helps the modular design of the runtime systems. These abstractions are increasing the role of the runtime system during execution. Runtime systems now perform complex scheduling and load balancing, orchestrate communication, drive accelerators and asynchronously execute graphs with billion tasks. In order for the runtime systems to perform as an active entity during execution, dynamic decision making and adaptation have become crucial. The trend suggests future HPC runtimes will perform more activities and provide even more abstractions.

References

[1] "Top 500 supercomputers published at sc20," https://www.top500.org/, accessed: 2021-01-01.

[2] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[3] S. Panneerselvam and M. Swift, "Rinnegan: Efficient resource use in heterogeneous architectures," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 2016, pp. 373–386.

[4] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. V. Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. P. Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme heterogeneity 2018 - productive computational science in the era of extreme heterogeneity: Report for DOE ASCR workshop on extreme heterogeneity," USDOE Office of Science (SC) (United States), Tech. Rep., 2018.

[5] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 69, 2015.

[6] Q. Zhu, B. Wu, X. Shen, L. Shen, and Z. Wang, "Co-run scheduling with power cap on integrated cpu-gpu systems," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 967–977.

[7] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "The complexity and approximation of optimal job co-scheduling on chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 7, 2011.

[8] A. Branover, D. Foley, and M. Steinman, "Amd fusion apu: Llano," *Ieee Micro*, vol. 32, no. 2, pp. 28–37, 2012.

[9] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter, "The tradeoffs of fused memory hierarchies in heterogeneous computing architectures," in *Proceedings of the 9th conference on Computing Frontiers*. ACM, 2012, pp. 103–112.

[10] "Mpich," https://www.mpich.org/, accessed: 2020-11-01.

[11] OpenMP, "OpenMP reference," 1999.

[12] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[13] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," in *ACM Sigplan Notices*, vol. 28, no. 10. ACM, 1993, pp. 91–108.

[14] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[15] NVidia, "CUDAZone," 2008.

[16] "Hip documentation," https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html, accessed: 2020-11-01.

[17] "Charm++: Programming language manual," http://charm.cs.uiuc.edu/~phil/manual.pdf, accessed: 2020-11-01.

[18] M. A. S. Bari, N. Chaimov, A. M. Malik, K. A. Huck, B. Chapman, A. D. Malony, and O. Sarood, "Arcs: Adaptive runtime configuration selection for power-constrained openmp applications," in *2016 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 2016, pp. 461–470.

[19] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *International Workshop on OpenMP*. Springer, 2013, pp. 171–185.

[20] "Programming model," https://en.wikipedia.org/wiki/Programming_model, accessed: 2020-11-01.

[21] "Execution model," https://en.wikipedia.org/wiki/Execution_model, accessed: 2020-11-01.

[22] "Openmp main site," https://www.openmp.org, accessed: 2020-11-01.

[23] "Hpx: Programming language manual," http://stellar.cct.lsu.edu/files/hpx-0.9.9/html/index.html, accessed: 2020-11-01.

[24] "Runtime ystem," https://en.wikipedia.org/wiki/Runtime_system, accessed: 2020-11-01.

[25] "The website of chapel parallel programming language," https://chapel-lang.org/, accessed: 2020-11-01.

[26] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[27] "History of supercomputing," http://wotug.org/parallel/documents/misc/timeline/timeline.txt, accessed: 2020-11-01.

[28] "Parallel computing works," http://www.netlib.org/utk/lsi/pcwLSI/text/BOOK.html, accessed: 2020-11-01.

[29] C. L. Seitz, "The cosmic cube," *Communications of the ACM*, vol. 28, no. 1, pp. 22–33, 1985.

[30] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, "The eden system: A technical review," *IEEE Transactions on Software Engineering*, no. 1, pp. 43–59, 1985.

[31] N. P. Kronenberg, H. M. Levy, and W. D. Strecker, "Vaxcluster: A closely-coupled distributed system," *ACM Transactions on Computer Systems (TOCS)*, vol. 4, no. 2, pp. 130–146, 1986.

[32] P. Pierce, "The nx/2 operating system," in *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues-Volume 1*, 1988, pp. 384–390.

[33] J. Flower and A. Kolawa, "Express is not just a message passing system current and future directions in express," *Parallel Computing*, vol. 20, no. 4, pp. 597–614, 1994.

[34] R. Butler and E. Lusk, "User's guide to the p4 programming system," Technical Report TM-ANL {92/17, Argonne National Laboratory, Tech. Rep., 1992.

[35] V. Bala and S. Kipnis, "Process groups: a mechanism for the coordination of and communication among processes in the venus collective communication library," in *[1993] Proceedings Seventh International Parallel Processing Symposium*. IEEE, 1993, pp. 614–620.

[36] J. Dongarra, G. Geist, R. Manchek, and V. S. Sunderam, "Integrated pvm framework supports heterogeneous network computing," *Computers in physics*, vol. 7, no. 2, pp. 166–175, 1993.

[37] W. Gropp and B. Smith, "Users manual for the chameleon parallel programming tools," Argonne National Lab., IL (United States); Office of Naval Research . . . , Tech. Rep., 1993.

[38] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and K. Yelick, "Parallel programming in split-c," in *Supercomputing'93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. IEEE, 1993, pp. 262–273.

[39] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield, "The amber system: Parallel programming on a network of multiprocessors," in *Proceedings of the twelfth ACM symposium on Operating systems principles*, 1989, pp. 147–158.

[40] K. Feind, "Shared memory access (shmem) routines," *Cray Research*, vol. 53, 1995.

[41] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and

computation," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2, pp. 256–266, 1992.

[42] B. N. Bershad and M. J. Zekauskas, "Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors," 1991.

[43] E. Upchurch, P. L. Springer, M. Brodowicz, S. Brunett, and T. D. Gottschalk, "Performance analysis of blue gene/L using parallel discrete event simulation," in *Lecture Notes in Computer Science : High Performance Computing - HiPC 2003*, 2003, pp. 2–11.

[44] "Co-array fortran," https://bluewaters.ncsa.illinois.edu/caf, accessed: 2021-01-01.

[45] "Titanium," http://titanium.cs.berkeley.edu/, accessed: 2021-01-01.

[46] I. Grasso and S. Pellegrini, "Libwater: heterogeneous distributed computing made easy," *Proceedings of the 27th ...*, pp. 161–171, 2013.

[47] "The website of x10," http://x10-lang.org/, accessed: 2020-11-01.

[48] K. B. Wheeler, R. C. Murphy, D. Stark, and B. L. Chamberlain, "The chapel tasking layer over qthreads." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2011.

[49] "Cuda runtime api," https://docs.nvidia.com/cuda/cuda-runtime-api/index.html, accessed: 2020-11-01.

[50] "The opencl specification," https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html, accessed: 2020-11-01.

[51] OpenACC, "OpenACC: Directives for accelerators," 2015.

[52] "Documentaion of starpu," https://files.inria.fr/starpu/doc/starpu.pdf, accessed: 2020-11-01.

[53] "Intel cilk plus," https://www.cilkplus.org/, accessed: 2020-11-01.

[54] "Intel tbb," https://software.intel.com/content/www/us/en/develop/tools/threading-building-blocks.html, accessed: 2020-11-01.

[55] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[56] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[57] "Amd corporation. rocm, a new era in open gpu computing." https://rocmdocs.amd.com/en/latest/, accessed: 2020-11-01.

[58] "The high performance open community runtime (p-ocr)," https://hpc.pnl.gov/projects/POCR/, accessed: 2020-11-01.

[59] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2017.

[60] T. Sterling, M. Anderson, and M. Brodowicz, "A survey: runtime software systems for high performance computing," *Supercomputing Frontiers and Innovations*, vol. 4, no. 1, pp. 48–68, 2017.

[61] P. Thoman, K. Hasanov, K. Dichev, R. Iakymchuk, X. Aguilar, P. Gschwandtner, E. Laure, H. Jordan, P. Lemarinier, K. Katrinis *et al.*, "A taxonomy of task-based technologies for high-performance computing," in *Proceedings of International Conference Parallel Processing and Applied Mathematics (To appear) Google Scholar*, 2018.

[62] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *ACM SigPlan Notices*, vol. 30, no. 8, pp. 207–216, 1995.

[63] "The cilk project from mit," http://groups.csail.mit.edu/sct/wiki/index.php?title=The_Cilk_Project, accessed: 2020-11-01.

[64] A. D. Robison, "Composable parallel patterns with intel cilk plus," *Computing in Science & Engineering*, vol. 15, no. 2, pp. 66–71, 2013.

[65] T. Willhalm and N. Popovici, "Putting intel® threading building blocks to work," in *Proceedings of the 1st international workshop on Multicore software engineering*. ACM, 2008, pp. 3–4.

[66] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[67] ——, "OpenMP: : An industry-standard API for shared-memory programming," *IEEE Computational Science AND Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[68] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang, "A proposal for task parallelism in openmp," in *International Workshop on OpenMP*. Springer, 2007, pp. 1–12.

[69] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[70] "Ompss programming model," https://pm.bsc.es/ompss, accessed: 2020-11-01.

[71] T. Dallou and B. Juurlink, "Hardware-based task dependency resolution for the starss programming model," in *2012 41st International Conference on Parallel Processing Workshops*. IEEE, 2012, pp. 367–374.

[72] E. Ayguade, R. Badía, and J. Labarta, "Ompss and the nanos++ runtime."

[73] "Mercurium is a source-to-source compilation infrastructure," https://pm.bsc.es/mcxx, accessed: 2020-11-01.

[74] "The nanos++ runtime system," https://pm.bsc.es/nanox, accessed: 2020-11-01.

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

[76] "The qthread library," https://cs.sandia.gov/qthreads/, accessed: 2020-11-01.

[77] R. B. Brightwell and S. L. Olivier, "Qthreads: Run time library support for task parallel programming." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[78] H. C. Edwards, S. L. Olivier, J. W. Berry, G. E. Mackey, S. Rajamanickam, M. Wolf, K. Kim, and G. W. Stelle, "Hierarchical task-data parallelism using kokkos and qthreads." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[79] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni *et al.*, "Parallel programming with migratable objects: Charm++ in practice," in *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, 2014, pp. 647–658.

[80] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *2009 International Conference on Parallel Processing Workshops*. IEEE, 2009, pp. 394–401.

[81] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Acm Sigplan Notices*, vol. 40, no. 10. ACM, 2005, pp. 519–538.

[82] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 2012, p. 66.

[83] "The website of legion," https://legion.stanford.edu/, accessed: 2020-11-01.

[84] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 263–276.

[85] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee *et al.*, "The open community runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–7.

[86] "Ocr-vx - an alternative implementation of the open community runtime," https://www.univie.ac.at/ocr-vx/, accessed: 2020-11-01.

[87] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar, "Ocr: the open community runtime interface," *Technical report*, 2015.

[88] "Argobots: A lightweight low-level threading framework," https://www.argobots.org/, accessed: 2020-11-01.

[89] "Charm++: Documentations," https://charm.readthedocs.io/en/latest/, accessed: 2020-11-01.

[90] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji, "Bolt: Optimizing openmp parallel regions with user-level threads," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 29–42.

[91] "Xcalablemp," https://xcalablemp.org/, accessed: 2021-01-01.

[92] "Parsec website," https://icl.utk.edu/parsec/index.html, accessed: 2020-11-01.

[93] Q. Meng, A. Humphrey, and M. Berzins, "The uintah framework: A unified heterogeneous task scheduling and runtime system," in *2012*

SC Companion: High Performance Computing, Networking Storage and Analysis. IEEE, 2012, pp. 2441–2448.

[94] J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*. IEEE, 2000, pp. 33–41.

[95] "The uintah website," http://www.uintah.utah.edu/, accessed: 2020-11-01.

[96] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to upc and language specification," Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Tech. Rep., 1999.

[97] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "Upc++: a pgas extension for c++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1105–1114.

[98] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[99] R. Keryell, R. Reyes, and L. Howes, "Khronos sycl for opencl: a tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, 2015, pp. 1–1.

[100] P. Thoman, P. Salzmann, B. Cosenza, and T. Fahringer, "Celerity: High-level c++ for accelerator clusters," in *European Conference on Parallel Processing*. Springer, 2019, pp. 291–303.

[101] "The openacc application programming interface," https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf, accessed: 2020-11-01.

[102] N. Otterness and J. H. Anderson, "Amd gpus as an alternative to nvidia for supporting real-time workloads," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[103] W. Kim and M. Voss, "Multicore desktop programming with intel threading building blocks," *IEEE software*, vol. 28, no. 1, pp. 23–31, 2010.

[104] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in parsec: a data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, 2017, p. 6.

[105] "Berkeley upc - unified parallel c," https://upc.lbl.gov/, accessed: 2020-11-01.

[106] "Hpx: Website," https://stellar-group.org/libraries/hpx/docs/, accessed: 2020-11-01.

[107] M. E. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," Ph.D. dissertation, Stanford University, 2014.

[108] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: the new adventures of old x10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, 2011, pp. 51–61.

[109] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: A scalable locality-aware adaptive work-stealing scheduler," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.

[110] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat, and M. Takeuchi, "Glb: Lifeline-based global load balancing library in x10," in *Proceedings of the first workshop on Parallel programming for analytics applications*, 2014, pp. 31–40.

[111] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "Gpu scheduling on the nvidia tx2: Hidden details revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2017, pp. 104–115.

[112] "Profiling and tuning openacccode," http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517B-Monday-Programming-GPUs-OpenACC.pdf, accessed: 2020-11-01.

[113] J. Tompson and K. Schlachter, "An introduction to the opencl programming model," *Person Education*, vol. 49, p. 31, 2012.

[114] T. Patki, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. De Supinski, "Exploring hardware overprovisioning in power-constrained, high performance computing," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 173–182.

[115] "Nvidia management library (nvml)," https://developer.nvidia.com/nvidia-system-management-interface, accessed: 2020-11-01.

[116] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures," in *2012 41st International Conference on Parallel Processing*. IEEE, 2012, pp. 48–57.

[117] T. Komoda, S. Hayashi, T. Nakada, S. Miwa, and H. Nakamura, "Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, 2013, pp. 349–356.

[118] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin, "Power-efficient time-sensitive mapping in heterogeneous systems," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 23–32.

[119] O. Sarood and L. V. Kale, "A'cool'load balancer for parallel applications," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11.

[120] "Papi measurement for llc-dram traffic," https://groups.google.com/a/icl.utk.edu/g/ptools-perfapi/c/NxpY2loJg4M/m/ESyCCBwYAAAJ?pli=1, accessed: 2020-09-01.

[121] H. McCraw, J. Ralph, A. Danalis, and J. Dongarra, "Power monitoring with papi for extreme scale architectures and dataflow-based programming models," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 385–391.

[122] M.-A. Hermanns, N. T. Hjlem, M. Knobloch, K. Mohror, and M. Schulz, "Enabling callback-driven runtime introspection via mpi_t," in *Proceedings of the 25th European MPI Users' Group Meeting*, 2018, pp. 1–10.

[123] S. Ramesh, A. Mahéo, S. Shende, A. D. Malony, H. Subramoni, A. Ruhela, and D. K. D. Panda, "Mpi performance engineering with the mpi tool interface: the integration of mvapich and tau," *Parallel Computing*, vol. 77, pp. 19–37, 2018.

[124] H.-H. Su, D. Bonachea, A. Leko, H. Sherburne, M. Billingsley, and A. D. George, "Gasp! a standardized performance analysis tool interface for global address space programming models," in *International Workshop on Applied Parallel Computing*. Springer, 2006, pp. 450–459.

[125] "Cupti documenation," https://docs.nvidia.com/cupti/Cupti/index.html, accessed: 2020-11-01.

[126] "Library for collecting amd gpus: roc-profiler," https://github.com/ROCm-Developer-Tools/rocprofiler, accessed: 2020-11-01.

[127] N. Chaimov, S. Shende, and A. D. Malony, "Multi-platform sycl profiling with tau," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–2.

[128] K. A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, "An Autonomic Performance Environment For Exascale," *Supercomputing frontiers and innovations*, vol. 2, no. 3, pp. 49–66, 2015.

[129] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: Towards automated performance tuning," in *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 2002, pp. 44–44.

[130] Y. Sun, J. Lifflander, and L. V. Kalé, "Pics: a performance-analysis-based introspective control system to steer parallel applications," in *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2014, p. 5.

[131] A. Porterfield, R. Fowler, A. Mandal, D. O'Brien, S. Olivier, and M. Spiegel, "Adaptive scheduling using performance introspection," TR-12-02. RENCI, 2012. R: http://www. renci. org/technical-reports/tr-12 . . . , Tech. Rep., 2012.

[132] Q. Zhu, B. Wu, X. Shen, L. Shen, and Z. Wang, "Understanding co-run degradations on integrated heterogeneous processors," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2014, pp. 82–97.

[133] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms," in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2017, pp. 1–10.

[134] M. Damschen, F. Mueller, and J. Henkel, "Co-scheduling on fused cpu-gpu architectures with shared last level caches," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2337–2347, 2018.

[135] S. Rai and M. Chaudhuri, "Exploiting dynamic reuse probability to manage shared last-level caches in cpu-gpu heterogeneous processors," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 3.

[136] A. Pan and V. S. Pai, "Runtime-driven shared last-level cache management for task-parallel programs," in *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12.

[137] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena, "Evaluating the effect of last-level cache sharing on integrated gpu-cpu systems with heterogeneous applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[138] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai, "Managing shared last-level cache in a heterogeneous multicore processor," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 225–234.

[139] S.-Y. Lee and C.-J. Wu, "Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2017, pp. 43–53.

[140] M. Hill and V. J. Reddi, "Gables: A roofline model for mobile socs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 317–330.

[141] Y. Cho, F. Negele, S. Park, B. Egger, and T. R. Gross, "On-the-fly workload partitioning for integrated cpu/gpu architectures." in *PACT*, 2018, pp. 21–1.

[142] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "Finepar: Irregularity-aware fine-grained workload partitioning on integrated architectures," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017, pp. 27–38.

[143] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 273.

[144] R. Kaleem, R. Barik, T. Shpeisman, C. Hu, B. T. Lewis, and K. Pingali, "Adaptive heterogeneous scheduling for integrated gpus," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 2014, pp. 151–162.

[145] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen, "Understanding co-running behaviors on integrated cpu/gpu architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 905–918, 2016.

[146] M. A. H. Monil, M. E. Belviranli, S. Lee, J. S. Vetter, and A. D. Malony, "Mephesto: Modeling energy-performance in heterogeneous socs and their trade-offs," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 413–425.

[147] L. Yu, D. Li, S. Mittal, and J. S. Vetter, "Quantitatively modeling application resiliency with the data vulnerability factor," *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2014.

[148] I. B. Peng, R. Gioiosa, G. Kestor, J. S. Vetter, P. Cicotti, E. Laure, and S. Markidis, "Characterizing the performance benefit of hybrid memory system for HPC applications," *Parallel Computing*, vol. 76, pp. 57–69, 2018.

[149] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[150] T. Allen and R. Ge, "Characterizing power and performance of gpu memory access," in *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC)*. IEEE, 2016, pp. 46–53.

[151] S. Lee, J. S. Meredith, and J. S. Vetter, "COMPASS: A framework for automated performance modeling and prediction," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. Newport Beach, California, USA: ACM, 2015, pp. 405–414.

[152] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[153] S. Lee and J. S. Vetter, "OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. Vancouver: ACM, 2014.