A Survey: Runtime Software Systems for High Performance Computing

Thomas Sterling[1]***, Matthew Anderson***[1]***, Maciej Brodowicz***[1]

HPC system design and operation are challenged by requirements for significant advances in efficiency, scalability, productivity, and portability at the end of Moore's Law with approaching nano-scale technology. Conventional practices employ message-passing programming interfaces; sometimes combining thread-based shared memory interfaces such as OpenMP. While these methods they are principally course grained and statically scheduled. Yet, performance for many real-world applications yield efficiencies of less than 10% even though some benchmarks achieve 80% efficiency or better (e.g., HPL). To address these challenges, strategies employing runtime software systems are being pursued to exploit information about the status of the application and the system hardware operation throughout the execution to guide task scheduling and resource management for dynamic adaptive control. Runtimes provide adaptive means to reduce the effects of starvation, latency, overhead, and contention. Many share common properties such as multi-tasking either preemptive or non-preemptive, message-driven computation such as active messages, sophisticated fine-grain synchronization such as dataflow and futures constructs, global name or address spaces, and control policies for optimizing task scheduling to address the uncertainty of asynchrony. This survey will identify key parameters and properties of modern and experimental runtime systems actively employed today and provide a detailed description, summary, and comparison within a shared space of dimensions. It is not the intent of this paper to determine which is better or worse but rather to provide sufficient detail to permit the reader to select among them according to individual need.

Keywords: runtime system, parallel computing, scalability, survey, HPC.

1. Introduction

A runtime system or just "runtime" is a software package that resides between the operating system (OS) and the application programming interface (API) and compiler. An instantiation of a runtime is dedicated to a given application execution. In this it is differentiated from the operating system in that the OS is responsible for the behavior of the full hardware system while the runtime is committed to some critical operational property of a specific user program. This important complementarity is realized by the OS that has information about the entire system workload and the objective function to optimize for, perhaps, highest possible job stream throughput while the runtime does not know about system-wide status. However, the runtime has direct information about the computational properties and requirements of its assigned application; information neither available nor actionable by the OS. This paper is a survey of runtime software systems being developed and employed for high performance computing (HPC) to improve the efficiency and scalability of supercomputers, at least for important classes of end-user applications.

In the most general sense, runtimes have been used in one form or another for more than five decades. In many cases these have served to close the semantic gap by exporting a virtual machine to the user greatly improving productivity and portability. Examples of such runtime system software include, but are in no way limited to, LISP, Basic, Smalltalk, and more recently Java and Python. With perhaps one important exception, very limited use of runtime systems has been made in the arena of HPC. This is not absolutely the case as some modest amount

[1]Center for Research in Extreme Scale Technologies, Indiana University

of runtime control has been employed even for the widely used programming interfaces of both OpenMP and MPI. But these uses are limited to the necessary essentials for the very good reason of avoiding software overheads. As will be discussed in more detail, the significant exception is the venerable Charm++ runtime software package that has been evolving for more than two decades with an emphasis on dynamic applications, in particular molecular dynamics.

There is a renewed, diverse, and expanding interest internationally in the development and application of advanced runtime systems for enabling progress in high performance computing with specific focus on applications employing dynamic adaptive behaviors and the range of emerging Petaflops scale computing systems approaching the realm of 1 Exaflops. The objectives of the body of work associated with this direction are:

- Efficiency – the gap between the best a system can do and the delivered performance on a real world application reflects the systems efficiency for that purpose. HPC operation may be improved by making runtime decisions informed by program and system state throughout the computation.
- Scalability – increasing the exposed and exploited parallelism may provide more computational work that can be done simultaneously either increasing throughput with more applied resources and possibly reducing the time to solution for a fixed size problem.
- User Productivity – much of the programming burden on the HPC user is due to the need to explicitly control the task scheduling and resource management by hand. Ideally, the user should be responsible for delineating properties of the problem algorithm leaving workflow management and distribution to some other mechanism such as a runtime system.
- Performance Portability – optimization of computing is conventionally highly sensitive to the particulars of HPC system architectures requiring changes, sometimes significant, as a parallel application is attempted to be ported to different architecture classes, scales, and generations. Runtime systems may be able to undertake the challenge of matching the needs of the applications with the particular capabilities and costs of each system to which it is to run.

Interestingly, although there are many characteristics among current runtime systems that are shared, they differ in origins and driving motivations. Different research drivers have included specific applications that require dynamic control and may benefit from adaptive real time operation, to support advanced programming languages and interfaces to provide higher level user semantics, to explore innovations in parallel computer architecture, for domain specific purposes, or to enable the exploitation of medium grain directed acyclic graph (DAG) representation of parallelism. In spite of these differentiated catalysts, many commonalities of approach emerge.

The strategy of employing runtime software for HPC is not obvious, nor assured. Early evidence suggests that some applications built with conventional programming methods that exhibit regular structures, uniform computation, and static data distribution and scheduling are unlikely to benefit from the use of runtimes. Because runtime software actually impose additional overhead to the overall computation, if sufficient benefit is not derived, the overheads may actually degrade the resulting performance. So, why can a runtime work? The opportunity is to result in superior task scheduling and resources management through exploitation of continuing runtime information combined with introspective policies for dynamic control.

In the following section, an extended discussion is presented to identify the fundamental factors that are driving early work in runtime software, both development and application.

These are related to evolving enabling technology trends, resulting architectures, and future application requirements. Section 3 presents the key areas of functionality that in one form or another is associated with many of the promising runtime systems as well as the ways that they vary. Section 4 provides detailed descriptions of a set of exemplar runtime systems noteworthy for their sophistication, their comprehensive capabilities, their breadth of application, and their leadership in the field. Within this select set are only those runtimes that scale across many nodes. For a more complete list, Section 5 delivers brief descriptions of a larger number of runtime including some limited to single node SMPs like OpenMP. To summarize as well as compare and contrast these runtimes, tables are provided in Section 6. Finally, this paper concludes by considering the future directions for further evolution and improvements of next generation runtime systems and their impacts on both end user applications and the architectures upon which they will run.

2. Drivers for HPC Runtime Systems

Conventional practices over the last two decades or more have proven very successful with exponential technology growth consistent with Moore's Law and corresponding progress in performance gain as reflected, for example, by dramatic improvements in measured Linpack [15] performance with a gain of 100,000× demonstrated in the last 20 years. Additional factors contributing to this are in the areas of parallel architecture and application algorithms. But in spite of these apparent successes, significant changes in technology trends have altered the means by which extended gains can be achieved. These changes are driving the need for runtime system development in support of many classes of applications.

Efficiencies which are measured as the ratio of sustained to peak performance may appear very high for certain favorable benchmarks, but for many real world applications at high scale the efficiency can be very much lower. Linpack Rmax values have been measured with optimized codes to in excess of 90% with even typical commodity clusters using Ethernet interconnect exceeding well beyond 60%. Routinely however, sophisticated and complex applications on large scale systems will often deliver sustained efficiency of less than 10% with even lower values below 5% not uncommon. As long as architecture design is structured to maximize ALU throughput, these metrics unsustainable if further significant performance growth within practical operational constraints is to be viable.

A wide array of computational challenges had been met with parallel algorithms that were coarse-grained, incorporated regular and often dense matrix data structures, were SPMD program flow controlled, and employed global barriers for synchronization. Many applications are now far more sophisticated than this combining irregular and time-varying data structures with medium to fine grained tasks to expose an abundance of parallelism for greater scalability. The evolution of the application execution is often unpredictable as it is sensitive to intermediate result data. Adaptive mesh refinement and n-body problems are only two of a large array of applications that fall into this category. As the underlying physical models increase in dimensionality and speciation for greater accuracy, non-linear sensitivities are exposed yet further complicating the control flow and inhibiting predictability for task scheduling and resource allocation at time of compile and launch.

HPC system structures have gone through rapid growth in scale, complexity, and heterogeneity, in particular, with the advent of multi-core and GPU accelerated architectures. These trends have increased since 2004 with the advent of multicore chips and their use in HPC systems

with such systems as the IBM BG/L. The largest supercomputer today, TaihuLight, comprises on the order of ten million cores. Both memory hierarchies and communication interconnection networks integrating all of the component subsystems are expanding aggravating the complexity of resource allocation, data distribution, and task scheduling. An important aspect of system behavior resulting from this is the uncertainty of asynchrony that is becoming worse. It means that both the access and service times local and remote across a large scale system can be dramatically different from each other and that for any single access may change in time over time. These factors clearly demand responsive measures to be included within system control. It has become clear with the emergence of modern high end machines that there is no single formula of system architecture but rather a plethora of choices with at least two emerging as front runners causing major challenges in portability. Fine grained architectures such as the early IBM Roadrunner, the IBM Blue Gene family of systems, the Chinese TaihuLight, and the Intel Knights Landing and next generation Knights Hill (such as the planned Aurora at ANL for 2018 operation) are all examples of this class of system. Alternatively, a design that mixes high speed general purpose processor cores for fast Amdahl code sequences with GPU accelerators for important numeric computing idioms exhibiting highly parallel fine grain dataflow is emerging as a second design point. The US's largest system, Titan at ORNL, is of this class as will be the follow-on machine there, Summit in 2018, that will include the IBM Power9. Other system classes are possible as well. Portability is greatly challenged with the need for computations to adapt for the diversity of architecture forms, scales, and future generations.

Pushing the edge of evolution of HPC system architecture to ever greater scales and diverse forms are fundamental technology trends that have changed dramatically from previous decades where the principal means of enhancing performance gain was by riding the exponential growth of Moore's Law for semiconductor device density. As silicon fabrication and manufacturing is converging at nano-scale, Moore's Law is flat-lining with little margins for marked improvements in the future. This does not mean that one time improvements in technology will not occur, but the dependence on exponential growth is ending. This follows the loss of Dennard scaling [14] for cores including the end of instruction level parallelism which ultimately proved a disappointment. Further constraints are imposed by having hit the power wall where the amount of energy that can be consumed is also bounded. Therefore, for greater capability, improved performance through enhanced efficiency and higher scalability will have to be derived with the future use of runtime systems as a possible important contributor. Runtimes will improve efficiency, at least for important classes of application algorithms, and can assist in exploiting a wider range of parallelism including through parallelism discovery from meta-data such as graph structures and DAG parallelism.

3. Key Functionality

Runtime systems for HPC vary widely in the specific details of their semantics and software implementation. But there has proven to be a convergence in general characteristics, although not complete uniformity. In the broadest sense, the functionality of a given runtime is a tradeoff between the richness of the semantics for ease of use and the effectiveness with which specific mechanisms can be implemented for purposes of efficiency and scalability. Runtime systems will also be distinguished by how they address a number of key operational factors. This section describes a number of such functionality factors that are likely to be found as components of more than one major runtime software package. One similarity among the majority of runtimes

is that they are a work in progress; even the oldest ones. Implementations are being constantly updated to take advantage of the latest developments in hardware and software environments, changes to user interfaces and intermediate forms to facilitate interoperability, and new policies, services, and advanced functional capabilities as experience dictates.

3.1. Multi-Threading

Almost all runtime systems in use have some form of threaded execution, whether directly employing POSIX Pthreads [30] of the host operating system or providing its own lightweight user threads package. The motivation is to provide a medium grained form of parallelism both to improve efficiency and to extend scalability. Static bulk synchronous parallel (BSP [38]) methods tend to leave some large gaps in the effective usage of physical computing resources in part as a result of the fork-join parallel control flow structures and because of static mapping of tasks to work units (e.g., cores). Many runtimes use medium grain threads in combination with dynamic scheduling to fill these gaps as becomes possible. This is enabled often by the use of over-decomposition and finer granularity when overheads permit. Some runtimes (e.g., ETI SWARM [16]) have ensured non-preemptive operation such that a thread once scheduled to a hardware compute resource goes to completion, thus permitting certain execution optimizations. Others permit preemption to avoid non-blocking due to requests to remote data accesses and services. The internal structure of flow control within a thread is usually sequentially consistent with some intra-thread ILP provided by the compiler and the hardware out of order execution completion. Typically, this is the final manifestation of the executable. But other versions include more complex intra-thread representations. Where the thread is generalized as a task, then DAG may be used to better represent parallelism of actions by reflecting control dependencies. Threads may differ by how they are instantiated. Value passing is popular where only input values determine a point of departure for starting in the style of dataflow. This can be a form of event-driven computation. How the thread interrelates with other threads and yet other forms of actions is in part determined by object naming conventions and synchronization semantics (see below).

3.2. Name Spaces and Addressing

Runtimes can differ significantly in the way they exchange information and intermediate results. The two extremes are pure distributed memory semantics and value-oriented data transfer on the one hand, and global shared memory semantics on the other. Some runtimes will only work on a SMP (symmetric multiprocessor) platform with hardware support for shared memory (e.g., Cilk++, OpenMP) which exhibits bounded time to completion and efficient address translation (e.g., TLB). Others provide greater scalability by taking advantage of multi-node DSM (distributed shared memory) systems by adapting to the asynchrony that causes variability of access time. Yet other runtimes assume a hardware structure exporting local shared memory within nodes with distributed memory mechanisms between nodes frequently with the equivalent of put/get semantics. That distinguishes between local and remote accesses at the programming model but still provides a form of global IDs. This allows programmers to explicitly control application use of locality. Finally, some modes maintain load/store semantics, both local and remote. These reduce user burden but require additional runtime mechanisms for efficient exploitation of locality to reduce latency effects.

3.3. Message-Driven Computing

Certainly not required of a runtime system but nonetheless frequently employed is the use of message-driven computation rather than exclusively message-passing as found with conventional CSP strategies. Message-driven computation moves work to the data rather than always requiring that data be gathered to the work. In the modern age, this is often referred to as "active messages" [39] taken from the very good and long-lasting work at UC Berkeley. But in truth the concepts go back to the 1970s with the dataflow model of computation and the Actors model [19] of computation. It is manifest in the experimental J-machine of the early 1990s and is implicit in the remote procedure calls of loosely coupled computing and in the cloud as well as transactional processing. So there is a long tradition and many forms of message-driven computation with as almost as many terms (e.g., 'parcels' for HPX runtime) for it as well. Its principal effect is to reduce the average latency of access and in combination with context-switched multi-threading to provide latency hiding. But more subtle effects are possible. One of the more important among these is the migration of continuations and the creation of a higher level of control state as will be discussed briefly in the next section. This permits the redistribution of the relationship of the execution on data and the control state that manages it for dynamic load balancing.

A packet of information that serves message-driven computation can and has taken many forms. Very simple formats are associated with dataflow tokens that merely have to carry a scalar datum or pointer to a predefined 'template' that itself specifies both the action to be performed and the destination of its own result values. The token/template relationship determines a control DAG that manages the information flow, exposes fine grain parallelism, and manages out of order computation and asynchrony. Unfortunately, direct implementations of this structure imposed too much overhead to be directly efficient as an architecture but the semantic concepts are important still. Expanded notions of message-driven computation include the specification of actions to be performed within the message itself, the destination as a virtual object, complex operand values structures or sets, and possible information about resulting continuations.

3.4. Synchronization

The flow control guiding the execution of the application with the involvement of a runtime system requires representation of the constraints to further progress. Such semantics of synchronization can be as coarse-grained as global barriers such as in the use of BSP or as fine grained as in the case of binary (two-operand) dataflow control. Runtime systems are taking two middle forms of synchronization in many cases that provide a richer semantics for small amount of overhead. One form of control is event driven synchronization to support DAG organized task-based parallelism. This is used in Parsec and OmpSs among other runtimes. This is similar to dataflow but not limited to value oriented arguments and fine grained control, thus able to amortize the overheads incurred.

Futures [5] based synchronization derived from the early Actors model extends the need for a result to an unbounded set of follow on actions. Using either eager or lazy evaluation the equivalent of an IOU is provided when an access request is made but the sought for value has yet to be determined. This is useful when the variable is manipulated within the meta-data of a structure such a graph but the actual value is not used. This exposes more parallelism for greater efficiency and scalability.

4. Major Runtime Exemplars

4.1. Charm++

Charm++ [23, 29] is a portable runtime system developed in the Parallel Programming Laboratory at University of Illinois at Urbana-Champaign by a diverse team directed by Laxmikant Kale. It targets primarily distributed memory systems and provides working implementations for clusters of compute nodes or workstations, IBM BlueGene L, P and Q families, and Cray XT, XE, and XC, but may also be used on isolated multi- and single-core machines as well as platforms equipped with accelerators such as GPUs and Cell BE. The remote communication layer natively supports InfiniBand, Ethernet, Myrinet, IBM, and Cray Gemini interconnects using protocols based on TCP/IP packets, UDP datagrams, IB verbs or vendor-specific LAPI, PAMI, and GNI interfaces. Charm++ can also utilize MPI [26] libraries if available on the specific platform. The runtime system accommodates a broad range of operating systems, ranging from Linux, Mac OS X, MS Windows, Cygwin UNIX emulation layer on Windows, through IBM AIX and CNK. While the native programming interface is expressed mainly in C++, binding with C and Fortan codes is also supported. Depending on the platform, CHARM++ programs may be compiled using GNU and Intel C++ compilers, clang, PGI compilers, IBM XL C, and MS Visual C++. The flagship applications include molecular dynamics package NAMD (shown to scale beyond 500,000 cores), quantum chemistry computation OpenAtom, and collision-less N-body simulator ChaNGa. The current revision of Charm++ software is v6.7.1.

The state in Charm++ programs is encapsulated in a number of objects, or chares. The objects communicate through invocation of entry methods. These methods may be called asynchronously and remotely. The runtime system distributes the chares across the parallel execution resources and schedules the invocations of their entry methods. The execution of Charm++ programs is message-driven, reminiscent of active-messages or Actors model. Both object instantiation as well as method call is mediated through proxy objects, or lightweight handles representing remote chares. Typically, the invocation of entry method on a chare proceeds to completion without an interruption. To avoid issues related to multiple concurrent updates to objects state, only one method is allowed to be active at a time. An executing method may call one or more methods on other chares. The chares are created dynamically at locations determined by dynamic load balancing strategy, although explicit specification of the destination processor is also possible. Chares may migrate to other locations during program run time by utilizing specialized migration constructor; migrated objects are then deleted from the original locations. Entry methods support arbitrary type signatures as defined by C++ language; however, they may not return values (the return type is void) thus permitting non-blocking operation. The method arguments are marshalled into messages for remote communication using PUP serialization layer. A more efficient direct invocation path for local objects that dispenses with proxy access is also supported by deriving local virtual addresses of target chares. While the preferred execution model is the structured dagger (SDAG) delineated above, Charm++ also provides thread-like semantics using its own implementation of user-level threads. In threaded mode, processing of a method can be suspended and resumed, multiple threads may be synchronized using, for example, futures, and methods may return a limited set of values. Since usage of global variables is not supported, Charm++ provides read-only variables that are broadcast to available processing elements after program begins to execute.

To facilitate the management of multiple objects, Charm++ distinguishes three types of chare collections: arrays, groups, and nodegroups. The first is a distributed array in which chares may be indexed by integers, tuples, bit vectors or user-defined types in multiple dimensions. There are several modes of element mapping to execution resources, including round-robin, block, block-cyclic, etc. in addition to user specified distributions. Arrays support broadcast (invocation of a method over all array elements) and reduction operations using a number of arithmetic, logic, and bitwise operators. Groups map each component chare onto individual processing elements. Analogously, chares belonging to a nodegroup are assigned to individual processes (logical nodes). Since a method invoked on a nodegroup may be executed by any processor available to the encompassing process, this may lead to data races as the concurrent execution of the same method on different processors is not prohibited. Charm++ offers exclusive entry methods for nodegroups if this behavior is undesirable. A dedicated chare (mainchare) is used to initiate the program execution. One or more mainchares are instantiated on processing element zero and create other singleton chares or chare collections as required, and initialize the read-only variables. Every chare and chare collection has a unique identifier providing a notion of global address space.

A Charm++ program is developed using standard C++ compiler and programming environment. Program source consists of header file, implementation file, and an interface file. The first two contain nominal C++ prototype definitions describing class members and inheritance from appropriate chare base classes (.h extension), and body code of entry methods (.cpp extension). The interface description (.ci extension) groups chares into one or more modules (which may be nested) and provides declarations of read-only variables, chare types, and prototypes of their entry methods (along with the required attributes, such as threaded or exclusive). The users compile the programs using charmc compiler wrapper which is also used to parse the interface description files and link the object files and runtime libraries into executable binaries. The Charm++ distribution includes a debugging tool, charmdebug, that may be used to record and scan execution traces, inspect memory, object and message contents, and freeze and reactivate selected sections of parallel programs. Also available is a Java-based performance visualization tool projections that displays the collected trace data showing interval-bound CPU utilization graphs, communication characteristics, per-core usage profiles, and histograms of selected events.

4.2. OCR

The Open Community Runtime (OCR) [36] is developed through collaboration between the Rice University, Intel Corporation, UIUC, UCSD, Reservoir Labs, Eqware, ET International, University of Delaware, and several national laboratories. The support for the project was also provided by the DoE Xstack program and DARPA UHPC program. The OCR effort focuses on development of the asynchronous task-based, resilient runtime system that targets exascale systems, but can be used on conventional single- and multi-node platforms. A number of proxy applications and numerical kernels, such as CoMD (molecular dynamics), LULESH (shock propagation), Cholesky matrix factorization, Smith Waterman algorithm (protein sequence comparison), 1D and 2D stencil codes, FFT, and triangle (recursive game tree search) have been ported to OCR while several others are work in progress. Source code of OCR is written in C permitting interfacing with all compatible languages and is distributed under BSD open source license. The supported processor targets include x86, Xeon Phi, and a strawman

Intel TG exascale architecture as well as different communication environments, such as MPI and GASNet. OCR is still in active development phase; the revision of its latest release is v1.1.0.

OCR exposes to the programmers three levels of abstraction in global namespace: data abstraction, compute abstraction, and synchronization abstraction. The data abstraction includes explicitly created and destroyed *data blocks* (DB). Each data block contains a fixed amount of storage space. Data blocks may migrate across the physical storage resources. Since conventional memory allocation wrappers, such as malloc, cannot be directly used for that purpose, data blocks are the only construct to provide dynamically allocated global memory for application data. The compute abstraction is built on top of *event-driven tasks* (EDTs). Such tasks represent fine-grain computation units in the program and are executed only when precedent input data blocks are ready. They may require zero or more scalar input parameters, zero or more input dependencies, and produce at most one output event. The output event is satisfied when the task execution completes. The event-driven tasks may also create other tasks and data blocks, however the creating EDT may not access the data blocks it created to prevent resiliency issues. The execution of any OCR program starts with the creation of a `mainEDT` task which initializes data blocks and spawns additional tasks as needed. The synchronization abstraction is used to define dependencies between executing tasks. It relies on *events* that act as a connector between producer and consumer EDTs. The information between the involved tasks is passed in data blocks with the exception of null identifier case that signifies pure synchronization without data propagation. Finally, the global namespace employs Globally Unique Identifiers (GUIDs) to track the instances of data blocks, event-driven tasks, and events in use on the machine. Additionally, GUIDs may identify task *templates*, or forms of task prototype that are consulted when creating new tasks. For synchronization, the GUID of a relevant event must be known in advance to the involved endpoint EDTs.

An arbitrary number of tasks may read from a data block, however, due to lack of explicit locking mechanisms in OCR, synchronization of exclusive writes is more involved. EDTs may gain access to data only in the beginning of execution, but can provide data at any moment. Typically, a given event may be satisfied only once and is automatically deleted after all dependent on it tasks are satisfied. In many cases, the common mechanism to accomplish synchronization relies on creation of additional EDTs and intermediate events. To enable more complex synchronization patterns that may be useful in handling race conditions, OCR provides other event types that differ in life cycle and semantics. Events of *sticky* type are retained until explicitly destroyed, typically by the data acquiring task. *Idempotent* events are similar to the sticky events, but subsequent attempts to satisfy them after the first one are ignored. A *latch* event contains two input slots and a trigger rule that defines condition when it becomes active; latch events are automatically deleted once triggered.

While the programmer may utilize the OCR interface directly, its intent is to provide low-level mechanisms for implementation of more sophisticated APIs. One of them is the CnC [10] (Intel Concurrent Collections) framework which allows the programmer to specify dependencies between the executing program entities as a graph by using a custom language. A CnC program applies *step collections* that represent computation to *item collections* representing data. The framework provides a translator to generate an OCR-CnC project containing Makefile, entry, exit, and graph initialization functions, skeletons of step functions, and glue code interacting with OCR. The user then has to fill in the provided function skeletons and compile the project to executables. Other programming flows are also possible and involve Hierarchically Tiled

Arrays (HTA) model [17], R-Stream compiler [25], Habanero-C [37], and Habanero-UPC++ [24] libraries.

4.3. HPX+

The family of HPX runtimes share a common inspiration of the ParalleX execution model. Different implementations have been developed to explore separate domains of interest and to serve possibly different sectors of the HPC community. The original HPX runtime [35], developed at Louisiana State University, has focused on C++ programming interfaces and has had a direct and recognized impact on the C++ steering committee and the emerging C++ programming standard. The HPX-5 runtime software [34] has been developed at Indiana University to serve as an interface to specific dynamic adaptive applications, some sponsored by DOE and NSF. HPX-5 has also been employed as an experimental platform to determine overhead costs, derive advanced introspective policies for dynamic adaptive control, and explore possible hardware architecture design to dramatically reduce overheads and increase parallelism. HPX+ which is captured in the table below is a work-in-progress with improved policy interface, real-time execution, automatic energy optimization, and fault tolerance as well as support for user debugging. Not all these features are fully functional at the time of writing but are in preparation for release. Although there are key distinctions between HPX and other runtimes described in this report, it subsumes most of the primary functionality.

HPX+ is an advanced runtime system software package developed as a first reduction to practice and proof of concept prototype of the ParalleX execution model. Its design goal is to dramatically improve efficiency and scalability through exploitation of runtime information of system and application state for dynamic adaptive resource management and task scheduling while exposing increased parallelism. HPX+ extends the core of HPX-5 to integrate advanced functionality associated with the practical concerns of fault tolerance, real-time operation, and energy management in conjunction with active user computational steering and in situ visualization. HPX+ is based on an extended threaded model that includes intra-thread dataflow operation precedent constraint specification. These threads are preemptive with fine-grained sequences that can be specified as non-preemptive for atomic operation. Threads as implemented in HPX+ are ephemeral and are first-class in that they are named in the same way as typed global data objects. They are also migratable; that is they can be moved physically while retaining the same virtual names. Message-driven computation is a key element of the strategy embodied by HPX+ with the parcel messaging model implemented on top of the innovative lightweight packets Photon system area network protocol with put-with-complete synchronization. Parcels target virtual objects, specify actions to be performed, carry argument data to remote locations, and determine continuations to be executed upon completion of the required action. Parcels may support remote global accesses and broader system wide data movement, instantiate threads either locally or at remote sites, directly change global control state, or cause I/O functions. HPX+ is a global address space system with a unique hierarchical abstraction layer referred to as ParalleX processes or Pprocesses. Pprocesses are contexts in which data, mappings, executing threads, and child Pprocesses are organized. They differ from conventional processes in a number of ways. They are first class objects and ephemeral like threads. But unlike threads (or other processes) they may span multiple system nodes, share nodes among processes (not limited to space partitioning) and migrate across the physical system, adding, deleting or swapping physical nodes. Further, Pprocesses may instantiate child processes. The

global address space gives load/store access to first class data anywhere in the ancestry hierarchy of the naming tree up to the root node or down to the deepest direct descendent. With cousin Pprocesses, accesses can only be achieved with access-rights and by method calls. Synchronization is achieved principally through dataflow and futures to manage asynchrony, expose parallelism, and handle both eager and lazy evaluation. A large global graph comprising futures at the vertices comprises an additional global control state that manages overall parallel operation including distributed atomic control operations (DCO) and copy semantics. Heterogeneous computing is supported through percolation, a variation of parcels that moves work to alternate physical resources such as GPUs.

4.4. Legion

Legion [32] is a programming model for heterogeneous distributed machines developed at Stanford University with contributions from Los Alamos National Laboratory and NVIDIA. It is motivated by the need for performance portability between machine architectures which differ substantially one from another and the need for programmability in an era when increases in component heterogeneity continue to complicate machine programmability. Legion targets machines with heterogeneous components that may result in larger component time variabilities than would be observed in a machine comprised of identical subcomponents.

The Legion programming model centers on three abstractions: tasks, regions, and mapping. A task is a unit of parallel execution. A region has an index space and fields that are frequently referred to as a collection of rows (index space) and columns (fields). Tasks works on regions and declare how they use their regions. Regions may have read, write, or reduction permissions. The mapper decides where tasks run and where regions are placed. The act of mapping consists of assigning tasks to a CPU or GPU and assigning regions to one of the appropriate memories accessible to the corresponding task. Both tasks and regions can be broken into subtasks or subregions, respectively. The Legion runtime system extracts parallelism by scheduling independent subtasks concurrently in this way: the runtime will automatically detect ordering dependencies in an application and create a dependency graph thereby allowing the runtime scheduler to concurrently schedule tasks as appropriate. The dependency graph is built by the runtime system and Legion schedules the graph dynamically in conjunction with the mapper while the programmer only writes the regions and subtasks for an application.

While generic task dependence analysis is a key component of Legion in order to extract parallelism and better utilize available compute resources, the application developer can also break the default sequential semantic behavior of subtasks on a region so that they become atomic or even simultaneous on a region and thereby enable uncontrolled reads or writes. Additional primitives such as acquire, release, make local copies, and phase barriers further refine this behavior and enable the application developer to bypass the generic dependence analysis in the runtime. Tasks on a critical path can be scheduled with priority and the mapping, while computed dynamically, can be controlled by the application to incorporate application specific knowledge.

Legion provides a C++ interface as well as its own programming language, Regent. When running on distributed heterogeneous machines, Legion uses GASNet [6] and CUDA and is still in active development. Legion has demonstrated scalability in combustion simulations using the S3D mini-application [18] on thousands of nodes of the largest machine in the United States, Titan. Legion provides several debugging tools for generating and viewing the task graph gener-

ated by the runtime from an application. Optimization of an application when moving between different machine architectures does not require that the programmer change the regions and subtasks already written. Only the mapping has to change when moving code between different machine architectures in order to optimize performance. Mapping changes such as changing where a task runs or where regions are placed do not affect the correctness of an application, only its performance.

Not unsurprisingly, the dynamic analysis of the application task dependency graph does add overhead in Legion that conventional programming models would not have. Legion is able to hide the additional overhead of the dynamic analysis by building the dependency graph sufficiently far ahead of the execution thereby making scheduling decisions well before execution allowing it to build up work in reserve and deal with high latency and asynchrony of operation by immediately using compute resources as they become available.

4.5. OmpSs

OmpSs [9] is a task based programming model developed at the Barcelona Supercomputing Center that aims address the needs of portability and programmability for both homogeneous and heterogeneous machines. The name originates from a combination of the OpenMP and the Star SuperScalar [33] programming model names. A runtime implementation of OmpSs is provided through the Nanos++ [8] runtime system research tool which provides user-level threads, synchronization support, and heterogeneity support and through the Mercurium compiler [7] for handling clauses and directives. OmpSs is often described either as an extension to OpenMP or as forerunner for OpenMP on emerging architectures, reflecting its aim to support asynchronous task based parallelism while also incorporating support for heterogeneous architectures in the same programming model. Like OpenMP, it can be used in a hybrid manner with MPI for simulations on distributed memory architectures.

The key abstractions in OmpSs are tasks with data-dependencies expressed through clauses and for-loops. Tasks in OmpSs are independent units of parallel execution. Parallelism is extracted through concurrent scheduling of tasks by the runtime system. For-loops operate in almost the same way in OmpSs as in OpenMP except with additional ability to alter the execution order that is useful for priority scheduling.

Because of the close relationship of OmpSs with OpenMP and because of widespread community familiarity with OpenMP, it can be helpful to compare and contrast OmpSs with OpenMP. Like OpenMP, OmpSs codes will execute correctly even if the OmpSs directives are ignored. Like OpenMP, OmpSs directives are also supplied as pragmas. However, unlike OpenMP, OmpSs does not have parallel regions and does not follow the fork-join execution model but rather implicitly extracts parallelism through concurrent scheduling of tasks. Launching OmpSs results in launching multiple user-level threads to which are assigned tasks when a task construct is encountered. The actual execution of the tasks will depend upon the task scheduling, thread availability, and resource availability.

Data dependencies can alter the order in which a task might execute and these data dependencies are expressed by adding clauses to the task construct. These clauses, including in, out, and inout, aid the runtime in creating a data dependence graph for use in scheduling the tasks. Additionally, a concurrent clause enables the concurrent execution with other similarly labeled tasks and places responsibility of uncontrolled read/writes on the programmer but also provides greater flexibility from using the task dependency graph generated by the runtime for

execution. OmpSs also provides key directives for task switching where execution on one task may switch to another. While this would normally occur when a task completes execution, the programmer may also force this to occur using the taskyield and taskwait directives. The task switching directive taskwait in OmpSs is frequently used in connection with a task reduction operation to wait on all child tasks to complete the reduction.

OmpSs is in active development and has a long track record of influencing the development of OpenMP. Many of the features available in OmpSs have been incorporated into the newest OpenMP specifications.

4.6. PaRSEC

The Parallel Runtime Scheduling and Execution Controller (PaRSEC) framework [21] is a task based runtime system which targets fine-grained tasks on distributed heterogeneous architectures. Like other runtime system frameworks, it aims to address the issues of portability across many hardware architectures as well as achieving high efficiency for a wide variety of computational science algorithms. Dependencies between tasks in PaRSEC are represented symbolically using a domain specific language called the Parameterized Task Graph [12]. The application developer can use the PaRSEC compiler to translate code written with sequential semantics like that of the SMPSs task-based shared memory programming model into a DAG and allow the runtime system to discover parallelism through dynamic execution. In this way, the runtime system is naturally well suited for heterogeneous architectures where a high variability in system component response time may be expected.

The principal abstractions of PaRSEc are tasks and data. A task in PaRSEC is a parallel execution unit with input and output data while data itself is a logical unit in the dataflow description. The Parameterized Task Graph resulting from the PaRSEC compiler is effectively a compressed representation of the task DAG and is fixed at compile time; it cannot represent dynamic DAGs nor data dependent DAGs. However, the Parameterized Task Graph provides an efficient symbolic representation of the task graph that eliminates the need for traversing the entire graph when making scheduling decisions. A local copy of the entire task graph is maintained on each node to avoid extra communication. Unlike many other task based runtime systems, PaRSEC maintains a task scheduler on each core. A user-level thread on each core alternates executing the scheduler to find new work or executing a task.

PaRSEC is developed at the Innovative Computing Laboratory at the University of Tennessee. It is still in active development with the most recent major release in 2015. Among the notable projects using PaRSEC is DPLASMA [20], an interface to ScaLAPACK using PaRSEC, and the open source high-performance computational chemistry tool NWCHEM 6.5 where a portion of the tool has been implemented with ParSEC capability [13]. PaRSEC also features a resilience policy with data logging and checkpointing integrated into the runtime model.

5. Other Runtimes

5.1. Qthreads

Sandia's Qthreads library [40] provides a distributed runtime system based on user-level threads. The threads are cooperatively scheduled and have small stacks, around 4-8 KB. The threads are transparently grouped to "shepherds" that refer to specific physical execution re-

sources, memory regions or protection domains. Association with shepherds identifies the location of thread execution. The threads are spawned explicitly and may synchronize their operation through mutexes and full-empty bits (FEBs). While mutexes may only be locked and unlocked, FEBs support explicit setting of FEB state (to full or empty) in a non-blocking fashion and blocking reads and writes to FEB-protected memory cells. The reads block on FEB until the contents is provided; the subsequent read of memory contents may clear or retain the full bit status of related FEB. Analogously, the writes come in two variants: one that blocks until the target FEB becomes empty and one that does not. A variant of thread called "future" permits the user to limit the total number of thread instances to conserve the system resources. A number of convenience constructs, such as parallel threaded loops and reduction operations are also provided. The remote operation is built on top of Portals4 library [31]. Qthreads execute on POSIX-compliant machines and have been tested on Linux, Solaris, and Mac OS using GNU, Intel, PGI, and Tilera compilers. The library has been ported to multiple CPU architectures, including x86, PowerPC, IA-64, and MIPS.

5.2. DARMA

The Distributed Asynchronous Resilient Models and Applications (DARMA) co-design programming model [41] is not a runtime system in itself but rather a translation layer between front end that provides a straightforward API for application developers based on key functionalities in asynchronous multitasking (AMT) runtime systems and a back end that is an existing AMT runtime system. Consequently, an application developer can write a DARMA code and run it using several different runtime system back ends as well as explore and improve the frontend DARMA API to reflect needs from the application developer community. Current back ends in DARMA include Charm++, Pthreads, HPX, and OCR with more back ends under development.

5.3. OpenMP and OpenACC

OpenMP [2], along with the OpenACC [1] extension targeting accelerators, is a directive based shared memory programming model utilizing concurrent OS threads. Both OpenMP and OpenACC require a suitable compiler of C, C++ or Fortran language that parses and converts to executable code segments of source suitably annotated by `#pragma` directives (or properly formed comments in Fortran). The primary focus of the implementation is simplicity and portability, hence not all semantic constructs provided by other multithreading runtime systems are supported. The model effectively subdivides the program code into sequentially executed and parallel sections. The latter are launched after implicit *fork* operation that creates or assigns a sufficient number of threads to perform the work. Parallel section is followed by an implicit *join* point which serves as a synchronization before proceeding to the following sequential region of the code. As the model does not specify communication environment, execution of OpenMP enabled programs on a distributed platform typically requires third party networking library, such as MPI. The compiler transformations applied to program code most commonly distribute iterations of a possibly mutiple-nested loop across the available execution resources (processor or GPU cores). The user also has a possibility to declare individual variables accessed in the loop scope as shared, private (with several variants) or reduction variables. The interface permits atomic, critical section, and barrier synchronization across the executing threads. One

of the recent additions was explicit task support. It is expected that OpenMP and OpenACC specifications will merge in not too distant future.

5.4. Cilk

The Cilk [27] family with its variants Cilk++ and Cilk Plus [22] is another compiler-driven approach to extracting task-level parallelism from application code. At its simplest, this is controlled by just two keywords, `spawn` and `join`, that respectively inform the compiler about possibility of instantiation of new parallel task(s) and enforce a logical barrier waiting for completion of all previously spawned tasks. The scheduler decides whether computations identified by spawn are offloaded to another OS thread or are continued in a different invocation frame by the calling thread. Cilk++ by Cilk Arts extended the use of fork-join model to C++. Cilk Plus, created by Intel after acquisition of Cilk Arts, introduced support for parallel loops and hyperobjects that enable multiple tasks to share the computational state without race conditions and need for locking. Common application of hyperobjects are monoid (reduction) operations. Cilk Plus functionality is supported by recent revisions of proprietary Intel C++ compiler, and open-source efforts GCC and Clang.

5.5. TBB

Intel Threading Building Blocks [11] is a C++ template library supporting generation and scheduling of multiple fine-grain tasks on shared memory multi-core platforms. The primary scheduling algorithm is work-stealing coupled with uniform initial distribution of computations across the available cores. TBB supports a wide range of parallel algorithms, including for loop, for-each loop, scan, reduce, pipeline, sort, and invoke. They may be applied to a number of concurrent containers, such as queues, hash maps, unordered maps and sets, and vectors. Synchronization primitives comprise basic atomic functions, locks, mutexes, and equivalents of C++11 scoped locking and condition variables. Task groups may be defined and dynamically extended for concurrent execution of specific tasks. Both blocking and continuation-passing task invocation styles are supported.

5.6. Argobots

Argobots [3] is a tasking runtime system that both supports concurrent task execution with dynamic scheduling and message-driven execution. Argobots is developed at Argonne National Laboratory and is well integrated with many of the other runtime systems reviewed here, including Charm++, PaRSEC, and OmpSs. Argobots interoperates with MPI and can be used as the threading model for OpenMP. An example of this is the BOLT implementation of OpenMP [4] which utilizes Argobots to provide OpenMP performance specialized for fine-grained application execution.

5.7. XcalableMP

XcalableMP (XMP) [42] targets distributed memory architectures with user specified directives to enable parallelization. These directives are intended to be simple and require minimal modifications to the original source code to enable parallelization similar to OpenMP but for distributed memory architectures. It incorporates a Partitioned Global Address Space (PGAS)

model and is influenced in design by High Performance Fortran. XMP also supports explicit message passing. An extension of XMP for heterogeneous clusters is under development known as XcalableACC. XMP and XcalableACC are part of the Omni compiler project [28] at RIKEN and the University of Tsukuba.

6. Summary Table

Table 1. Comparison of primary runtime system properties.

Runtime	Distributed	Task dependency based	Scheduler	Heterogeneous	Programming interface	Preemptive	Thread support
Argobots	Y	N	FIFO	N	C	Y	Y
Charm++	Y	N	structured DAG (FIFO)	N	C++, custom	N	Y
Cilk Plus	N	N	work-stealing	N	C, C++	N	N
DARMA	Y	N	dynamic, dependency based	Y	C++	Y	Y
HPX+	Y	N	FIFO, work-stealing	Y	C, C++	N	Y
Legion	Y	Y	dynamic, dependency based	Y	Regent, C++	N	Y
OCR	Y	Y	FIFO, work-stealing, priority work-sharing, heuristic	N	C	N	N
OmpSs	N*	Y	dynamic, dependency based	Y	C, C++, Fortran	Y	Y
OpenMP OpenACC	N	N	static, dynamic, dependency based	Y	C, C++, Fortran	OS level	Y
PaRSEC	Y	Y	static, dynamic, dependency based	Y	C, Fortran	Y	Y
Qthreads	Y	N	FIFO, LIFO, centralized queue, work-stealing, and others	N	C	N	Y
TBB	N	N	LIFO, work-stealing, continuation-passing style, affinity, work-sharing	N	C++	N	Y
XcalableMP	Y	N	static, dynamic	N**	C, Fortran	Y	Y

* The COMPSs project is the programming model implementation from the StarSs family intended for distributed computing.

** The XcalableACC project is corresponding project for heterogeneous computing.

Table 2. Comparison of primary runtime system properties (continued).

Runtime	Priority queue	Message-driven	GAS support	Synchronization	Task DAG support	Logical hierarchical namespace	Termination detection
Argobots	Y	Y	N	locks, mutexes	N	N	Y
Charm++	message prioritization	Y	Y	actors model, futures	Y	N	Y
Cilk Plus	N	N	N	join, hyperobjects	N	N	N
DARMA	N	N	N	dependencies	Y	N	Y
HPX+	N	Y	Y	futures (local control objects)	Y	Y	Y
Legion	Y	N	N	coherence modes: exclusive, atomic, concurrent	Y	N	Y
OCR	Y	Y	Y	events, dependencies	Y	N	N
OmpSs	Y	N	N	events, dependencies	Y	N	Y
OpenMP OpenACC	Y	N	N	join, atomics, critical section, barrier	N	N	N
PaRSEC	Y	N	N	locks, mutexes, dependencies	Y	N	Y
Qthreads	Y	Y	N	FEB, mutex	N	N	N
TBB	3 level static & dynamic	N	N	locks, mutexes, atomics, condition variables	N	N	N
XcalableMP	N	N	Y	barrier, post, wait	N	N	Y

The key aspects of discussed runtime systems are compared in Table 1 and 2.

7. Conclusions and Future Work

Runtime system software packages are emerging as an augmenting way to possibly dramatically improve efficiency and scalability, at least for important classes of applications and hardware systems. There are a number of runtime systems at various stages of development and proven experience demonstrating a diversity of concepts and implementation methods. Many different choices are represented to satisfy a varying set of requirements including semantics of underlying execution models, integration with bindings to preferred programming models, exploitation of available libraries and other software packages, tradeoffs between overheads and parallelism, and policies for dynamic adaptive resource management and task scheduling. Yet, it is also apparent with closer scrutiny that although such runtimes are derived from different starting conditions they have converged to a certain degree exhibiting many common properties. This should be reassuring as it seems that similar answers are derived in spite of disparate initial

conditions. This does not mean that the community is agreed upon a single runtime semantics let alone syntax. But rather that there appears to be many likely conceptual elements that will contribute to one or a few selected runtimes. It is premature to standardize as more experience is required to identify best practices. But that process has begun.

Still there are significant obstacles. Among these are overheads, exposure of parallelism, load balancing, scheduling policies, and programming interfaces. These represent areas of future work. A particular challenging problem is integrating many node hardware systems into a single system image seamlessly without the intrinsically greater latencies of data movement dominating the effectiveness of parallel processing. Policies of load balancing that minimize long distance data transfers can mitigate this concern but must be closely associated with the specific details of the application parallel algorithms and the time-varying data structures associated with the evolving computations. If this cannot be resolved, it may demonstrate that prior practices engaging explicit programmer control may be necessary after all. On the other hand, it is found that some applications exhibit highly unpredictable behavior and data structures requires similar forms of adaptive control. Such applications like AMR and FMM to name a couple must include this dynamic control whether provided by a runtime system or by the programmer. Finally, in the long term, if runtimes are to prove broadly useful, they will require some architecture support as part of the hardware to minimize overheads, limit latencies, and expose as much parallelism as possible for scalability. Thus it is possible that the near term explorations of runtime software for optimizing applications will ultimately drive changes in the HPC hardware system design.

References

1. The OpenACC application programming interface, October 2015. Version 2.5, `http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf`.

2. OpenMP application programming interface, November 2015. Version 4.5, `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.

3. Argonne National Laboratory. Argobots: a lightweight low-level threading/tasking framework, Nov 2016. Version 1.0a1 `http://www.argobots.org/`.

4. Argonne National Laboratory. BOLT: a lightning-fast OpenMP implementation, Nov 2016. Version 1.0a1 `http://www.mcs.anl.gov/bolt/`.

5. Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. In *SIGART Bull.*, pages 55–59, New York, NY, USA, August 1977. ACM.

6. Berkeley Laboratories, University of California Berkeley. GASNet low-level networking layer, Oct 2016. Version 1.28.0, `https://gasnet.lbl.gov/`.

7. Barcelona Supercomputing Center. The Mercurium source-to-source compilation infrastructure, June 2016. Version 2.0.0 `https://pm.bsc.es/mcxx`.

8. Barcelona Supercomputing Center. The Nanos++ runtime system, June 2016. Version 0.10 `https://pm.bsc.es/nanox`.

9. Barcelona Supercomputing Center. The OmpSs programming model, June 2016. Version 16.06 https://pm.bsc.es/ompss.

10. Intel Corp. Intel® concurrent collections C++ API, June 2014. Website, https://icnc.github.io/api/index.html.

11. Intel Corp. Intel® Threading Building Blocks (Intel® TBB), 2017. Website, http://www.threadingbuildingblocks.org.

12. A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra. Ptg: An abstraction for unhindered parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 21–30, Nov 2014.

13. A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra. Parsec in practice: Optimizing a legacy chemistry application through distributed task-based execution. In *2015 IEEE International Conference on Cluster Computing*, pages 304–313, Sept 2015.

14. Robert H. Dennard, Fritz Gaensslen, Hwa-Nien Yu, Leo Rideout, Ernest Bassous, and Andre LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid State Circuits*, 9(5), October 1974.

15. Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation, Practice and Experience*, 15(9), July 2003.

16. ET International, Inc. SWARM (SWift Adaptive Runtime Machine), 2011. White paper, http://www.etinternational.com/files/2713/2128/2002/ETI-SWARM-whitepaper-11092011.pdf.

17. Basilio B. Fraguela, Ganesh Bikshandi, Jia Guo, María J. Garzarán, David Padua, and Christoph Von Praun. Optimization techniques for efficient HTA programs. *Parallel Comput.*, 38(9):465–484, September 2012.

18. R. Grout, R. Sankaran, J. Levesque, C. Woolley, S. Posy, and J. Chen. S3D direct numerical simulation: preparation for the 10-100 PF era, May 2012. http://on-demand.gputechconf.com/gtc/2012/presentations/S0625-GTC2012-S3D-Direct-Numerical.pdf.

19. C. Hewitt and H. G. Baker. Actors and continuous functionals. Technical report, Cambridge, MA, USA, 1978.

20. University of Tennessee Innovative Computing Laboratory. DPLASMA: distributed parallel linear algebra software for multicore architectures, April 2014. Version 1.2.0 http://icl.utk.edu/dplasma/.

21. University of Tennessee Innovative Computing Laboratory. The PaRSEC generic framework for architecture aware scheduling and management of micro-tasks, Dec 2015. Version 2.0.0 http://icl.cs.utk.edu/parsec/index.html.

22. Intel Corp. *Intel® Cilk™ Plus Language Specification*, 2010. Version 0.9, document number 324396-001US, https://www.cilkplus.org/sites/default/files/open_specifications/cilk_plus_language_specification_0_9.pdf.

23. L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

24. Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. HabaneroUPC++: A compiler-free PGAS library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 5:1–5:10, New York, NY, USA, 2014. ACM.

25. Richard Lethin, Allen Leung, Benoit Meister, and Eric Schweitz. R-stream: A parametric high level compiler, 2006. Reservoir Labs Inc., Talk abstract, http://www.ll.mit.edu/HPEC/agendas/proc06/Day2/21_Schweitz_Abstract.pdf.

26. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, June 2015. Specification document, http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

27. MIT. *Cilk 5.4.6 Reference Manual*, 1998. http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf.

28. Omni compiler project. Omni, Nov 2016. Version 1.1.0 http://omni-compiler.org/.

29. Parallel Programming Laboratory, University of Illinois at Urbana-Champaign. The Charm++ parallel programming system manual. Version 6.7.1, http://charm.cs.illinois.edu/manuals/pdf/charm++.pdf.

30. POSIX - Austin Joint Working Group. 1003.1-2008 – IEEE Standard for Information Technology – Portable Operating System Interface (POSIX®). IEEE Standard, 2008. http://standards.ieee.org/findstds/standard/1003.1-2008.html.

31. T. Schneider, T. Hoefler, R. E. Grant, B. W. Barrett, and R. Brightwell. Protocols for fully offloaded collective operations on accelerated network adapters. In *42nd International Conference on Parallel Processing*, pages 593–602, Oct 2013.

32. E. Slaughter, W. Lee, Z. Jia, T. Warszawski, A. Aiken, P. McCormick, C. Ferenbaugh, S. Gutierrez, K. Davis, G. Shipman, N. Watkins, M. Bauer, and S. Treichler. Legion programming system, Feb 2017. Version 16.10.0, http://legion.stanford.edu/.

33. Vladimir Suboti, Steffen Brinkmann, Vladimir Marjanovi, Rosa M. Badia, Jose Gracia, Christoph Niethammer, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Programmability and portability for exascale: Top down programming methodology and tools with starss. *Journal of Computational Science*, 4(6):450 – 456, 2013. Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

34. The Center for Research in Extreme Scale Technologies. HPX-5, Nov 2016. Version 4.0.0 http://hpx.crest.iu.edu/.

35. The Ste||ar group. HPX, July 2016. Version 0.9.99, http://stellar.cct.lsu.edu/.

36. Mattson Tim and Cledat Romain. OCR, the open community runtime interface, March 2016. Version 1.1.0, `https://xstack.exascale-tech.com/git/public?p=ocr.git;a=blob;f=ocr/spec/ocr-1.1.0.pdf`.

37. Rice University. Habanero-C, 2013. Website, `https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C`.

38. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

39. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. *Proceedings of The 19th Annual International Symposium on Computer Architecture, 1992*, pages 256–266, 1992.

40. Kyle Wheeler, Richard Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (MTAAP '08 workshop)*, 2008.

41. J. Wilke, D. Hollman, N. Slattengren, J. Lifflander, H. Kolla, F. Rizzi, K. Teranishi, and J. Bennett. DARMA 0.3.0-alpha specification, March 2016. Version 0.3.0-alpha, SANDIA Report SAND2016-5397.

42. XcalableMP specification working group. XcalableMP: a directive-based language extension for scalable and performance-aware parallel programming, Nov 2014. Version 1.2.1 `http://www.xcalablemp.org/`.