Supercomputing Frontiers and Innovations

2017, Vol. 4, No. 1

Scope

- Enabling technologies for high performance computing
- Future generation supercomputer architectures
- Extreme-scale concepts beyond conventional practices including exascale
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Distributed operating systems, kernels, supervisors, and virtualization for highly scalable computing
- Scalable runtime systems software
- Methods and means of supercomputer system management, administration, and monitoring
- Mass storage systems, protocols, and allocation
- Energy and power minimization for very large deployed computers
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Parallel performance and correctness debugging
- Scientific visualization for massive data and computing both external and in situ
- Education in high performance computing and computational science

Editorial Board

Contents

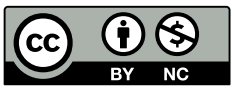
Design and Implementation of the PULSAR Programming System for Large Scale Computing
J. Kurzak, P. Luszczek, I. Yamazaki, Y. Robert, J. Dongarra 4

Workflows for Science: a Challenge when Facing the Convergence of HPC and Big Data
R. M. Badia, E. Ayguade, J. Labarta 27

A Survey: Runtime Software Systems for High Performance Computing
T. Sterling, M. Anderson, M. Brodowicz 48

xSDK Foundations: Toward an Extreme-scale Scientific Software Development Kit
R. Bartlett, I. Demeshko, T. Gamblin, G. Hammond, M. Heroux, J. Johnson, A. Klinvex, X. Li, L. C. McInnes, J. D. Moulton, D. Osei-Kuffuor, J. Sarich, B. Smith, J. Willenbring, U. M. Yang 69

Performance Portability of HPC Discovery Science Software: Fusion Energy Turbulence Simulations at Extreme Scale
W. Tang, B. Wang, S. Ethier, Z. Lin 83



DOI: 10.14529/jsfi170101

Design and Implementation of the PULSAR Programming System for Large Scale Computing

Jakub Kurzak[1]***, Piotr Luszczek***[1]***, Ichitaro Yamazaki***[1]***, Yves Robert***[2]***, Jack Dongarra***[1]

The objective of the PULSAR project was to design a programming model suitable for large-scale machines with complex memory hierarchies, and to deliver a prototype implementation of a runtime system supporting that model. PULSAR tackled the challenge by proposing a programming model based on systolic processing and virtualization. The PULSAR programming model is quite simple, with point-to-point channels as the main communication abstraction. The runtime implementation is very lightweight and fully distributed, and provides multithreading, message-passing and multi-GPU offload capabilities. Performance evaluation shows good scalability up to one thousand nodes with one thousand GPU accelerators.

Keywords: runtime scheduling, dataflow scheduling, distributed computing, massively parallel computing, multicore processors, hardware accelerators, virtualization, systolic arrays.

Introduction

Motivation

High-end supercomputers are on the steady path of growth in size and complexity. One can get a fairly reasonable picture of the road that lies ahead by examining the platforms that will be brought online under the DOEs CORAL initiative. In 2018, the DOE aims to deploy three different CORAL platforms, each over 150 petaflop peak performance level. Two systems, named Summit and Sierra, based on the IBM OpenPOWER platform with NVIDIA GPU-accelerators, were selected for Oak Ridge National Laboratory and Lawrence Livermore National Laboratory; an Intel system, based on the Xeon Phi platform and named Aurora, was selected for Argonne National Laboratory.

Summit and Sierra will follow the hybrid computing model, by coupling powerful latency-optimized processors with highly parallel throughput-optimized accelerators. They will rely on IBM Power9 CPUs, NVIDIA Volta GPUs, and NVIDIA NVLink interconnect to connect the hybrid devices within each node, and a Mellanox Dual-Rail EDR Infiniband interconnect to connect the nodes. The Aurora system, on the contrary, will offer a more homogeneous model by utilizing the Knights Hill Xeon Phi architecture, which, unlike the current Knights Corner model, will be a stand-alone processor and not a slot-in coprocessor, and will also include integrated Omni-Path communication fabric. All platforms will benefit from recent advances in 3D-stacked memory technology.

Overall, both types of systems promise major performance improvements: CPU memory bandwidth is expected to be between 200 GB/s and 300 GB/s using HMC; GPU memory bandwidth is expected to approach 1 TB/s using HBM; GPU memory capacity is expected to reach 60 GB (NVIDIA Volta); NVLink is expected to deliver no less than 80 GB/s, and possibly as high at 200 GB/s, of CPU to GPU bandwidth. In terms of computing power, the Knights Hill is expected to be between 3.6 teraFLOPS and 9 teraFLOPS, while the NVIDIA Volta is expected to be around 10 teraFLOPS.

[1]University of Tennessee, Knoxville, USA
[2]École Normale Supérieure de Lyon, Lyon, France

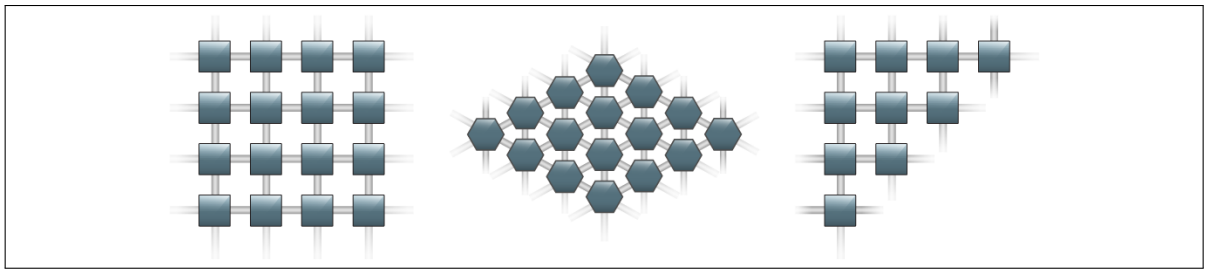


Figure 1. Canonical systolic array shapes

And yet, taking a wider perspective, the challenges are severe for software developers who have to extract performance from these systems. The hybrid computing model seems to be here to stay, and memory systems will become even more complicated. It is clear that support for parallelism is going to have to dramatically increase, going up by at least an order of magnitude for the CORAL systems and achieving billion-way parallelism at exascale. The PULSAR project attempts to tackle these challenges with a simple programming model, based on the systolic processing model, augmented with virtualization. The programming model is simple and so is the runtime implementation. Processing is completely distributed and, therefore, very scalable.

Background

Systolic arrays are descendants of array-like architectures such as iterative arrays, cellular automata and processor arrays. A systolic array is a network of processors that rhythmically compute and pass data through the system. The seminal paper by Kung and Leiserson [42] defines systolic arrays as devices with *"simple and regular geometries and data paths"* with *"pipelining as general methods of using these structures"*.

The systolic array paradigm is a departure from the von Neuman paradigm. While the von Neuman architecture is instruction-stream-driven by an instruction counter, the systolic array architecture is data-stream-driven by data counters. A systolic array is composed of matrix-like rows of units called cells or *Data Processing Units* (DPUs). DPUs operation is transport-triggered, i.e., triggered by the arrival of a data object. The DPUs are connected in a mesh-like topology (often two-dimensional). Each DPU is connected to a small number of nearest neighbor DPUs and performs a sequence of operations on data that flows through it. Often different data streams flow across the mesh in different directions. Figure 1 shows three canonical shapes of systolic arrays: *square* can be used for a dense matrix multiplication, *diamond* for a band matrix factorization, *triangle* for a dense matrix factorization.

Early on, Kung identified the main strength of systolic arrays as ability to addressing the problem of the I/O bottleneck: *"Thus, a problem what was originally compute-bound can become I/O-bound during its execution. This unfortunate situation is the result of a mismatch between the computation and the architecture. Systolic architectures, which ensure multiple computations per memory access, can speed up compute-bound computations without increasing I/O requirements"* [41]. Other valuable properties of systolic arrays include highly scalable parallelism, modularity, regularity, local interconnection, high degree of pipelining, and highly synchronized multiprocessing.

Closely related to systolic arrays is the concept of *wavefront arrays*, where global synchronization is replaced by dataflow principles. Wavefront arrays are derived by tracing computa-

tional wavefronts in the algorithm, and pipelining these wavefronts on the processor array. The computing network serves as a data-wave-propagating medium.

The computational wavefronts are data-driven. In a sense, they are similar to electromagnetic wavefronts, since each processor acts as a secondary source and is responsible for the activation of the next front. Despite the lack of global timing, the sequencing of tasks is correctly followed. Whenever data are available, the sender informs the receiver and the receiver accepts the data whenever required. This scheme can be implemented through a simple handshaking protocol, which ensures that the computational wavefronts propagate in an orderly manner. Wavefront arrays share the features of regularity, modularity, local interconnection, and pipelining: *"A wavefront array equals a systolic array plus dataflow computing"* [43].

Most importantly, computations expressed as a *Direct Acyclic Graph* (DAG) can be mapped to an array processor, by assigning multiple nodes of the DAG to each processing element, as long as the DAG is uniform (shift-invariant). Examples of algorithms, which belong to this class include: convolution, autoregressive filtering, discrete Fourier transforms and an array of dense linear algebra algorithms - matrix multiplication, LU factorization, QR factorization, triangular matrix inversion, and more.

The origins of systolic arrays can be traced back to *parallel array computers* such as the Solomon computer [29] and its successor ILLIAC IV [7, 40]. At the peak of interest in the mid 80s systolic arrays targeted special-purpose algorithm-oriented VLSI implementations, often as attached processors. The ideas also led to the design of the *Warp* machines [3], which were a series of increasingly general-purpose systolic array processors created by Carnegie Mellon University in conjunction with industrial partners: G. E., Honeywell and Intel, and funded by the U. S. *Defense Advances Research Projects Agency* (DARPA). Interest in systolic arrays died off by early 90s, mostly due to the high cost of implementing them as special-purpose hardware during a time in which Moore's law was relentlessly increasing the computing power and decreasing the cost of general-purpose processors.

In their seminal paper [42], Kung and Leiserson applied systolic arrays to problems in dense linear algebra: matrix multiplication, Gaussian elimination, and triangular solve. They also pointed out applications in signal processing: convolution, *Finite Impulse Response* (FIR) filter, and discrete Fourier transform. A large body of work on systolic arrays was devoted to applications in dense linear algebra [1, 6, 8, 17, 28, 46, 56].

Despite the loss of interest in systolic arrays per se, systolic principles lead to a series of efficient algorithms for general-purpose computer systems, the prime example being a series of algorithms for matrix multiplication, including: Cannon's [10], Fox's [26], BiMMeR [30], PUMMA [16], SUMMA [58], and DIMMA [15].

The paper by Fisher and Kung [24] offers an extensive overview of systolic array literature. General discussion and motivation for systolic arrays is given by Kung [41], and Fortes and Wah [25]. Systematic treatment of the topic is provided by Robert [52, 53] and Evans [22]. The paper by Kung [43] coins the term *wavefront arrays* and discusses the mapping of task graphs to systolic architectures. The paper by Johnsson et al. [32] talks about *general purpose* systolic arrays that can be applied to a wider range of problems (reconfigurable computing).

Related Work

The emerging Exascale programming models, including languages, draw from the PGAS (Partitioned Global Address Space) and APGAS (Asynchronous PGAS) efforts. These efforts

include the DARPA-sponsored HPCS (High Productivity Computing Systems) program [21] which stressed productivity rather than performance, with the latter being the prerequisite for the former. The older PGAS languages include Titanium [4, 19, 37–39, 44, 45, 50], UPC (Unified Parallel C) [11, 18], and CAF (Co-Array Fortran) [23, 51]. They use the concept of globally visible address space with an explicit handling of addresses that are non-local. The ongoing efforts in the Fortran community ensure continuous support for CAF functionality as exemplified by CAF 2.0 [31, 48, 49, 54] and incorporation of some of the CAF features into the Fortran 2008 standard [55]. The DARPA's HPCS program introduced three more languages into the space: Fortress [2, 33], X10 [59], and Chapel (Cascade High Productivity Language) [12, 13]. The last two are still maintained by IBM and Cray, respectively. A recent resurgence of activity around shared memory programming resulted in the OpenSHMEM [14] project that borrows some ideas from the PGAS model, but is much more library-centric, as opposed to requiring a completely new programming language.

The other approach to achieving good performance on the current Petascale and future Exascale hardware designs is to use software runtime systems. Two notable projects in this category are Charm++ [36] and PaRSEC [9], which deal with algorithms and their implementation represented as a Direct Acyclic Graph (DAG) of tasks connected with edges that communicate data between them – a concept clearly related to the dataflow paradigm. Many other systems offer similar paradigm but might not afford the same type of support for distributed memory parallelism [5, 47].

A new execution model has been argued by the authors of ParalleX [27, 34, 35] and implemented by the HPX project [57] that now serves as a clear need for extensions of the `C++` standard. Codelets Execution Model [20] can also be considered in the category of the new models of computation.

Outline

The rest of the paper is organized as follows. We describe the PULSAR programming model in Section 1, and further explain the construction and operation of a PULSAR instance in Section 2. We outline the runtime implementation in Section 3. Then the following sections are devoted to the detailed description of an example, namely Cannon's algorithm. We briefly review the algorithm in Section 5 and report performance results in Section 6. Finally, we state some final remarks in Section 6.

1. Programming Model

The PULSAR programming model relies on five abstractions to define the computation: *VSA*, *VDP*, *channel*, *packet*, *tuple*; and on two abstractions to map the computation to the actual hardware: *thread*, *device*. Figure 2 conveys the basic ideas.

Virtual Systolic Array (VSA) A set of VDPs connected with channels.

Virtual Data Processor (VDP) The basic processing element in the VSA.

Channel A point-to-point connection between a pair of VDPs.

Packet The basic unit of information transferred in a channel.

Tuple A unique VDP identifier.

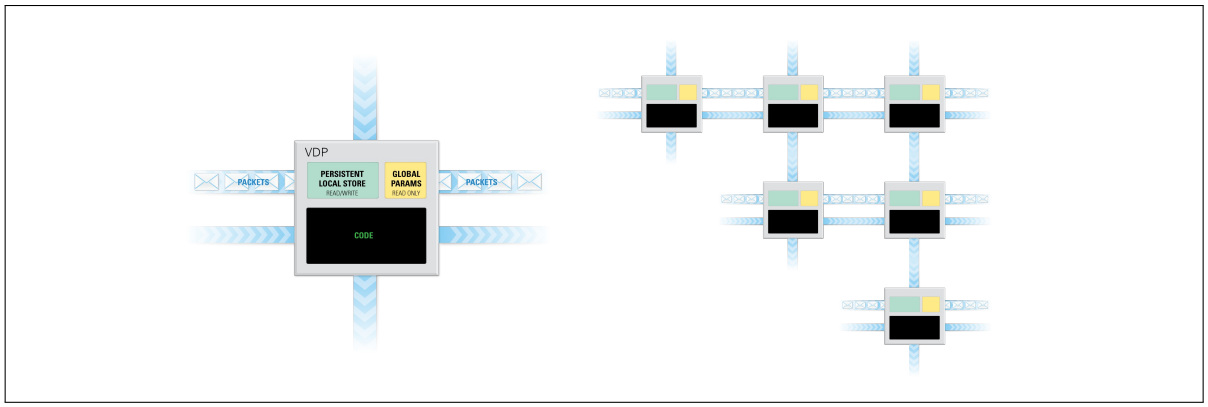


Figure 2. A VDP (left) and a VSA (right)

Thread Synonymous with a CPU thread or an entire (multicore) CPU.

Device Synonymous with an accelerator device (GPU, Xeon Phi, etc.)

The sections to follow describe the roles of the different entities, how the VDP operation is defined, how the VSA is constructed, and how the VSA is mapped to the hardware. These operations are accessible to the user through PULSAR's *Application Programming Interface* (API). Because this API is quite small (12 core functions and 6 auxiliary functions), actual function names are used when describing the different actions. Currently, PULSAR is implemented in C and exports C bindings.

1.1. Tuple

Tuples are strings of integers. Each VDP is uniquely identified by a tuple. Tuples can be of any length, and different length tuples can be used in the same VSA. Two tuples are identical if they are of the same lengths and have identical values of all components. Tuples are created using the variadic function `prt_tuple_new`, which takes a (variable length) list of integers as its input. The user only creates tuples; after creation, the tuples are passed to VDP and channel constructors. They are destroyed by the runtime at the appropriate time of destroying those objects. As a general rule in PULSAR, the user only creates objects and loses their ownership after passing them to the runtime.

1.2. Packet

Packets are basic units of information exchanged through channels connecting VDPs. A packet contains a reference to a continuous piece of memory of a given size. Conceptually, packets are created by VDPs. The user can use the VDP function `prt_vdp_packet_new` to create a new packet. A packet can be created from preallocated memory by providing the pointer. Alternatively, new memory can be allocated by providing a NULL pointer. The VDP can fetch a packet from an input channel using the `prt_vdp_channel_pop` function, and push a packet to an output channel using the `prt_vdp_channel_push` function. The `prt_vdp_packet_release` function can be used to discard a packet. This does not translate to immediate deallocation, since a packet can have multiple active references. The runtime discards a packet when the number of active references goes down to zero. The VDP does not lose the ownership of the

packet after pushing it to a channel. The packet can be used until the `prt_vdp_packet_release` function is called.

1.3. Channel

Channels are unidirectional point-to-point connections between VDPs, used to exchange packets. Each VDP has a set of input channels and a set of output channels. Packets can be fetched from input channels and pushed to output channels. Channels in each set are assigned consecutive numbers starting from zero (or *slots*). Channels are created using the `prt_channel_new` function and providing tuples of source and destination VDPs, and slot numbers in the source and destination VDPs. The user does not destroy channels. The runtime destroys channels at the time of destroying the VDP. After creation, the channel needs to be inserted in the appropriate VDP, using the `prt_vdp_channel_insert` function. The user needs to insert a full set of channels to each VDP. At the time of inserting the VDP in the VSA, the system joins channels that identify the same communication path.

1.4. VDP

The VDP is the basic processing element of the VSA. Each VDP is uniquely identified by a tuple. The VDP is assigned a function which defines its operation. Within that function, the VDP has access to a set of global parameters, its private, persistent local storage, and its channels. The runtime invokes that function when there are packets in all of the VDP's input channels. This is called firing. When the VDP fires, it can fetch packets from its input channels, call computational kernels, and push packets to its output channels. It is not required that these operations are invoked in any particular order. The VDP fires a prescribed number of times. When the VDP's counter goes down to zero, the VDP is destroyed. The VDP has access to its tuple and its counter.

At the time of the VDP creation, the user specifies if the VDP resides on a CPU or on an accelerator. This is an important distinction, because the code of a CPU VDP has synchronous semantics, while the code of an accelerator VDP has asynchronous semantics. In other words, for a CPU VDP, actions are executed as they are invoked, while for an accelerator VDP, actions are queued for execution after preceding actions complete. In the CUDA implementation, each VDP has its own stream. All kernel invocations have to be asynchronous calls, placed in the VDP's stream. Similarly, in the case of an accelerator VDP, all channel operations have asynchronous semantics.

1.5. VSA

The VSA contains all VDPs and their channel connections, and stores the information about the mapping of VDPs to the hardware. The VSA needs to be created first and then launched. An empty VSA is created using the `prt_vsa_new` function. Then VDPs can be inserted in the VSA using the `prt_vsa_vdp_insert` function. Then the VSA can be executed using the `prt_vsa_run` function, and later destroyed using the `prt_vsa_delete` function.

At the time of creation, using the `prt_vsa_new` function, the user provides the number of CPU threads to launch per each distributed memory node, and the number of accelerator devices to use per each node. The user also provides a function for mapping VDPs to threads,

```
prt_packet_t *packet = prt_vdp_packet_new(vdp, ...);
kernel_that_writes(..., packet->data, ...);
prt_vdp_channel_push(vdp, slot, packet);
prt_vdp_packet_release(vdp, packet);

prt_packet_t *packet = prt_vdp_channel_pop(vdp, slot);
kernel_that_modifies(..., packet->data, ...);
prt_vdp_channel_push(vdp, slot, packet);
prt_vdp_packet_release(vdp, packet);

prt_packet_t *packet = prt_vdp_channel_pop(vdp, slot);
prt_vdp_channel_push(vdp, slot, packet);
kernel_that_reads(..., packet->data, ...);
prt_vdp_packet_release(vdp, packet);
```

```
for (v = 0; v < vdps; v++) {
    prt_vdp_t *vdp = prt_vdp_new(...);
    for (in = 0; in < inputs; in++) {
        prt_channel_t *input = prt_channel_new(...);
        prt_vdp_channel_insert(vdp, input, ...);
    }
    for (out = 0; out < outputs; out++) {
        prt_channel_t *output = prt_channel_new(...);
        prt_vdp_channel_insert(vdp, output, ...)
    }
    prt_vsa_vdp_insert(vsa, vdp, ...);
}
```

Figure 3. Code snippets for VDP operation (left) and VSA construction (right)

and another for mapping VDPs to devices. These functions have to return the global thread or device number, based on the VDP's tuple and the total thread count or device count.

VSA construction can be replicated or distributed. The replicated construction is more straightforward from the user's perspective. In the replicated construction, each MPI process inserts all the VDPs, and the system filters out the ones that do not belong in a given node, based on the VDP-to-thread and the VDP-to-device function. However, the VSA construction process is inherently distributed, so each process can also insert only the VDPs that belong in that process.

2. Construction and Operation

The VSA is first constructed, and then launched. The VSA is constructed by creating all VDPs and inserting them in the VSA. Each VDP, in turn, is constructed by creating all its channels and inserting then in the VDP. The operation of the VSA is defined through the operation of its VDPs. VDPs operate by launching computational kernels, and communicating by writing and reading packets to and from their channels.

2.1. VSA Construction and Launching

Figure 3 shows a simple code snippet for VSA construction. A VSA is created using the `prt_vsa_new` function, which returns a new VSA with an empty set of VDPs. After creation, the VSA has to be populated with VDPs. Then the VSA can be launched using the `prt_vsa_run` function. After execution, the VSA can be destroyed using the `prt_vsa_delete` function. The function destroys all resources associated with the VSA.

2.2. VDP Creation and Insertion

Figure 3 shows simple code snippets of VDP operation. A VDP is created using the `prt_vdp_new` function. The function returns a pointer to a new VDP with empty sets of input and output channels. After creation, the VDP has to be populated with channels. Then the VDP can be inserted into the VSA using the `prt_vsa_vdp_insert` function. The user does not free the VDP. At the time of calling `prt_vsa_vdp_insert`, the runtime takes ownership of the VDP. The VDP will be destroyed in the process of the VSA execution or at the time of calling `prt_vsa_delete`.

The user has to define the VDP function. The runtime invokes that function when packets are available in all of the VDP channels, which is called *firing*. Inside that function, the user has access to the VDP object. In particular, the user has access to its tuple, counter, local store and global store. `global_store` is the read-only global storage area, passed to the VSA at the time of its creation. `local_store` is the VDP private local storage area, which is persistent between firings. `tuple` is the VDP unique tuple, assigned at the time of creation. `counter` is the VDP counter. At the first firing, the counter is equal to the value assigned at the time of the VDP creation. At each firing the counter is decremented by one. At the last firing the counter is equal one.

2.3. Channel Creation and Insertion

A channel is created using the `prt_channel_new` function. After creation, the channel can be inserted into the VDP using the `prt_vdp_channel_insert` function. The user does not free the channel. At the time of calling `prt_vdp_channel_insert`, the runtime takes ownership of the channel. The channel will be destroyed in the process of the VSA execution or at the time of calling `prt_vsa_delete`.

2.4. Mapping of VDPs to Threads and Devices

The user defines the placement of VDPs on CPUs and GPUs by providing the mapping function at the time of the VSA creation with `prt_vsa_new`. The runtime calls that function for each VDP and passes as parameters: the VDP tuple, the pointer to the global store, the total number of CPU threads at the VSA disposal in that launch, and the total number of devices in that launch.

The function has to return the mapping information in an object of type `prt_mapping_t`, with the fields `location` and `rank`, where the location can be either `PRT_LOCATION_HOST` or `PRT_LOCATION_DEVICE`, and the rank indicates the global rank of the unit.

2.5. VDP Operation

This section describes actions which can take place inside the VDP function, i.e., the function passed to `prt_vdp_new`. The user never calls that function; it is called by the runtime when packets are available in all active input channels of the VDP. Inside that function, computational kernel can be launched and packets can be created, deleted, pushed down output channels and fetched from input channels.

A new data packet can be created by calling the `prt_vdp_packet_new` function and released by calling the `prt_vdp_packet_release` function. The runtime will keep the packet and its data around until it completes all pending operations associated with the packet. However, the packet and its data should not be accessed after the release operation.

A packet can be received by calling the `prt_vdp_channel_pop` function, and sent by calling the `prt_vdp_channel_push` function. The packet is still available to the VDP after calling the send function and can be used and repeatedly sent until it is released.

2.6. Channel Deactivation and Reactivation

A channel can be deactivated by the VDP by calling the `prt_vdp_channel_off` function. This indicates to the runtime that it can schedule the VDP (call the VDP function) without checking if there are packets in that channel. The VDP should not attempt to read packets from an inactive channel.

A channel can be reactivated by the VDP by calling the `prt_vdp_channel_on` function. This indicates to the runtime that it cannot schedule the VDP (call the VDP function) if there are no incoming packets in that channel. By default, channels are active.

2.7. Handling of Tuples

A new tuple can be created by calling the variadic function `prt_tuple_new`. A tuple has to have at least one element. There is no upper limit on the number of elements. Tuples are dynamically allocated strings or integers, with the `INT_MAX` constant at the end, serving as the termination symbol. As such, tuples can be freed by calling the C standard library `free` function. However, tuples should not be freed after passing them to the PULSAR runtime. The runtime will free all such tuples during its operation or at the time of calling `prt_vsa_delete`.

3. Runtime Implementation

Figure 4 shows the main objects in the runtime implementation and their relations. The VSA is the top-level object containing multiple threads and devices, threads being synonymous with CPU cores, and devices being synonymous with accelerator devices. It also contains a single instance of the communication proxy, which is a server-like object responsible for managing inter-node (MPI) and intra-node (PCI) communication. Each thread and device maintains a list of multiple VDP. Each VDP maintains two separate lists of channels, one for input channels, one for output channels. Each channel maintains a list of packets.

In a CPU-only scenario, there may be no devices; in a GPU-only scenario, there may be no threads. Depending on the distribution of VDPs to threads and devices, any particular thread or device may end up with an empty list of VDPs. There may be VDPs with no input channels - pure data producers, as well as VDPs with no output channels - pure data consumers. In practical scenarios, most VDPs will have a number of input and output channels. A list of packets in a channel will grow and shrink at runtime.

3.1. Tuple

Tuples are implemented as strings of integers, terminated with the `INT_MAX` marker. Tuples support copy, concatenation and two types of comparisons. One checks for an exact match in size and content, the other implements lexicographical ordering.

3.2. Packet

A packet is the basic unit of data in PULSAR. It contains a pointer to a memory region and its size and the number of active references to the packet. A packet can be actively being used by a VDP while also residing in multiple channels. A VDP can keep using a packet after pushing

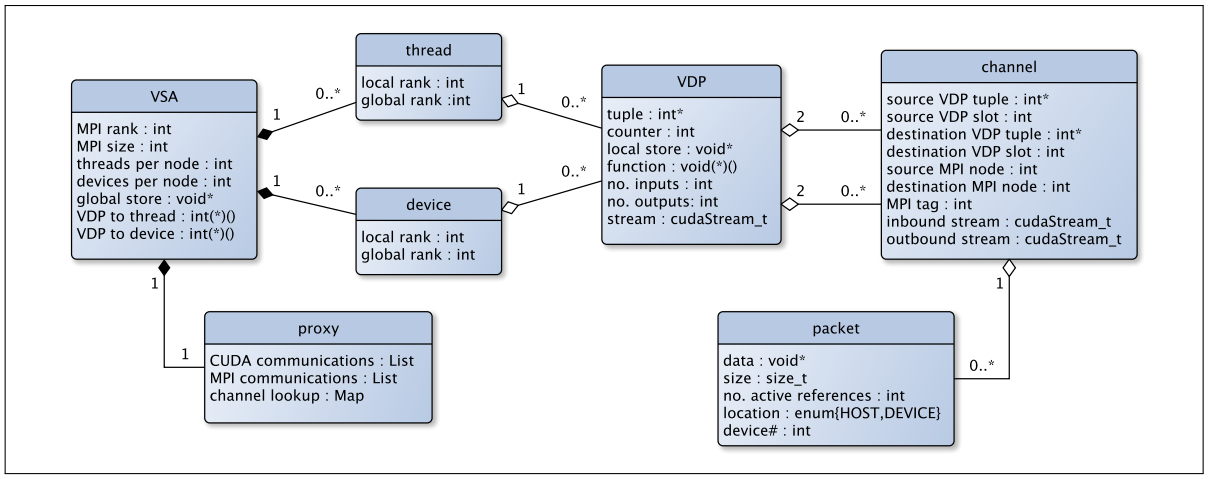


Figure 4. Structure of PULSAR runtime

it to a channel, or it can push it to multiple channels to implement a broadcast or multi-cast operation.

While the packet is a very simple concept in the context of a CPU implementation, support for accelerators introduces an additional level of complexity, due to the fact that accelerators have separate memories. A shared-memory system with multiple accelerators basically looks to the programmer like a distributed-memory, global address space system. To handle this situation, PULSAR runtime keeps track of the location of the packet, which can be either in the host (CPU) memory, or in the memory of one of the accelerators. This is the reason a packet is not a stand-alone object, but is subordinate to a VDP. Packets are created by VDPs and inherit their initial locations from the VDPs that create them.

Another level of complexity is introduced by the fact that PULSAR cannot rely on CUDA functions for device memory allocations, without sacrificing asynchronous semantics, since CUDA allocations cannot be executed in a stream. Because of that, PULSAR implements its own device memory allocator, which grabs a large chunk of the device memory at the time of initialization, and assigns memory segments asynchronously at runtime. Currently, the implementation is very simple, with fixed size (configurable) segments and fixed size (also configurable) initial reservation. PULSAR maintains one such allocator per device, and each packet maintains a reference to its allocator.

3.3. Channel

Channels are packet carriers between VDPs. A channel knows the tuples and slots of its source and destination VDPs, and maintains a list of packets. A channel also knows the numbers (MPI ranks) of the nodes, where the source and destination VDPs reside, as well as its unique tag for MPI communication between the pair of nodes it connects. A channel connecting to a device VDP is also assigned a unique stream to allow for asynchronous communication (one that does not block kernel launches).

A channel provides two services to the VDP, fetching a packet (the *pop* operation), and sending a packet (the *push* operation). A VDP is only fired when there are packets in all its active input channels. Therefore, the pop operation is trivial, and simply fetches a packet from the queue. All the complexity of communication is in the push operation, which takes appropriate actions, depending on the boundaries crossed by the channel. If the source and the destination

VDPs are both CPU VDPs, residing in the same node, the operation is as simple as moving the packet from one queue to another. Things are more complicated if the channel crosses node boundaries, in which case MPI is invoked. Yet a different scenario is implemented if a boundary between a host memory and a device memory has to be crossed. And finally, the most complex case is invoked when a packet residing in device memory is sent across the network. The last case results in a sequence of CUDA callbacks and non-blocking MPI calls, initiated by the channel and carried out by the communication proxy, with support from the CUDA runtime and MPI.

As an extension to the basic programming model, a channel contains an *active/inactive* flag, which allows the VDP for suspending channel activities. By deactivating a channel, the VDP pledges to not pop packets from that channel, which allows the VDP to fire with that channel being empty. Newly created channels are active by default. A VDP can deactivate and re-activate channels at will, as long as it does not attempt to fetch packets from an inactive channel.

3.4. VDP

A VDP is the basic execution unit of the VSA. A VDP knows its tuple and counter, and the lists of input and output channels. It is assigned a function, local store and global store. Support for accelerators requires a VDP to also know its location, i.e., if it is assigned to a CPU or an accelerator, and which accelerator, if there are multiple of them in a node. An accelerator VDP is also assigned its unique stream, so that multiple VDPs can execute at the same time, in different streams, and also kernel launches can overlap data transfers.

In order to hide the complexity of managing the memory system within a single node (host plus multiple devices), the VDP provides more abstract functions for accessing lower-level functions of packets and channels. Specifially, rather than calling packet and channel methods directly, the user calls VDP methods to perform operations on packets and channels. E.g., to create a packet, the user calls the VDP function `prt_vdp_packet_new`, so that the packet can inherit its initial location from the creator VDP. For consistency, a packet is released by the VDP upon calling the function `prt_vdp_packet_release`. Similarly, channels are accessed by VDP functions `prt_vdp_channel_push` and `prt_vdp_channel_pop`. First, this is consistent with the handling of packets. Second, the user designates channels by using their slot numbers, rather than references, which slightly increases the level of abstraction.

3.5. Threads and Devices

Threads and devices are internal PULSAR objects not directly accessed by the user. The user only deals with them indirectly by providing formulas for mapping of VDPs to threads and devices. Threads and devices know their local (within a node) and global ranks, and maintain lists of VDPs.

The main reason for the distinction between CPU threads and GPU devices is the synchronous behavior of the former and asynchronous behavior of the latter. While the code of a CPU VDP is expected to have the usual synchronous nature, the code of a GPU VDP is expected to be fully asynchronous. The reason is the asymmetry in the programming models for CPUs and GPUs, in which GPUs are not fully autonomous devices. Most of their actions have to be initiated by a CPU thread, and synchronous GPU code locks up the controlling thread. In a multi-GPU setup, this can be remedied by putting a separate CPU thread in charge

of controlling each GPU. This still leaves the problem with asynchronous communication and multi-stream execution.

The requirement for the GPU VDPs to only contain asynchronous calls allows for maximum performance and the simplest runtime implementation. A single CPU thread is sufficient to launch all the computation and communication to multiple GPUs, and in the actual PULSAR implementation, it is the same thread that is also responsible for all MPI transactions.

3.6. VSA

The VSA is the main object in PULSAR, containing all the top-level information about the system, including: the total number of nodes and the rank of the local node, the number of CPU threads launched per node, and the number of GPU devices used per node. It contains the lists of threads and devices, and the communication proxy. It also contains a number of auxiliary structures, like a lookup table of local VDPs, and a list of channels connecting the local node to other nodes, as well as the list of memory allocators for all local devices.

The most complicated function provided by the VSA is VDP insertion. First, the VSA evaluates the mapping function to find out the VDP location. VDPs that do not belong in the local node, are immediately discarded. VDPs that do belong in the local node, are inserted in the appropriate thread or device, depending on the location returned by the mapping function.

Then local channels are merged. Each VDP is inserted with a complete set of channels. If the newly inserted VDP has channels connecting to other VDPs, inserted before, each pair of duplicate channels is merged into one channel. To support this operation, the VSA maintains a hash table, where the VDPs can be looked up by their tuples.

Then a unique (MPI) tag is assigned to each channel going out of the node. All channels connecting each pair of nodes have consecutive tags. This is necessary, because PULSAR implements VDP-to-VDP communication on top of MPI. MPI messages are received with the `MPI_ANY_SOURCE, MPI_ANY_TAG` flags and the destination VDP identified by the rank of the origin and the tag of the channel. Consecutive numbering of tags connecting each pair of nodes, as opposed to, e.g., global numbering in the whole system, prevents the problem of exhausting the 16-bit tag size limit of older MPI implementations. To support this operation, the VSA maintains lists of channels connecting the local node to all other nodes.

Finally, at the time of inserting a device VDP, a unique stream is created and assigned to the VDP to enable its asynchronous operation.

At the time of launch, the VSA basically launches threads and waits for their completion. Each CPU thread carries out its own execution. For a CPU-only run, nothing more is required. If the run is distributed and/or accelerators are involved, the VSA launches an extra thread for the proxy, which carries out the firings of device VDPs, as well as the MPI and PCI communication.

3.7. Proxy

The proxy carries out all tasks of asynchronous nature, including communication and firing of device VDPs. For the purpose of communicating, threads and devices register with the proxy as agents. The proxy maintains a list of sends requested, per agent, and sends posted, per agent. It also maintains a list of outstanding receive requests, one for all agents, as well as a list of outstanding local transfers, also one for all agents. During execution, the proxy continuously loops over the following actions:

- Post another send for each agent.
- Complete another send for each agent.
- Post another receive.
- Complete another receive.
- Cycle each device, i.e., fire another VDP on each device.
- Issue all local communications.

The most complicated transfer that may happen in the system is when a GPU sends a packet across the network. Figure 5 illustrates that situation. The following sequence of events takes place:

1. In the VDP function, the `prt_vdp_channel_push` function is called do push a packet down a channel. At this point, the packet resides in device memory. A CUDA callback is placed in the VDP stream, to trigger the transfer, when all preceding VDP operations complete.
2. CUDA runtime executes the callback, which places the transfer request in proxy's list of local requests.
3. The proxy handles the transfer request by placing asynchronous device-to-host memory copy, across the PCI, in the outbound stream associated with the channel performing the communication.
4. The proxy follows up with placing a callback in the channel's outbound stream, to trigger a transfer across the network, when the PCI transfer completes.
5. CUDA runtime executes the callback, which places the transfer request in the proxy's list of local requests.
6. The proxy places a send request in the list of sends requested by the device.
7. The proxy issues the `MPI_Isend` action, and moves the request from the list of sends requested to the list of sends posted.
8. The proxy tests the send request for completion. When the request completes, the proxy removes the request from the list of sends posted and releases the packet.

A similar, although not identical, sequence of actions happens on the receiving side, where the message is received, the destination VDP identified, and appropriate steps taken depending on its location. Specifically, if the destination VDP resides in one of the devices, a transfer across the PCI is queued with the proxy. As long as the sequence is, there are no shortcuts here, since potentially two PCI buses and the network interface have to be crossed by the data. Little can be done about the latency of such transfers, however, all the emphasis in PULSAR is on latency hiding, rather than minimizing. The objective is to keep the buses and network interfaces saturated by multiple transfers going on at any given point in time.

4. Software Engineering

Although being an experimental project, PULSAR has a quite robust implementation. PULSAR is coded in C in a fairly object-oriented manner, and would be very suitable for a C++ implementation. PULSAR is very compact with only 21 .c files, 22 .h files and 6,600 lines of code. The compactness and clear structure is mostly thanks to the very crisp abstractions established in the design process (VDP, VSA, channel, packet, etc.)

PULSAR has very few software dependencies. At the minimum, it requires Pthreads, and can be optionally compiled with MPI support and/or CUDA support. The build process involves compiling the sources and creating a library from the object files. Using MPI requires providing

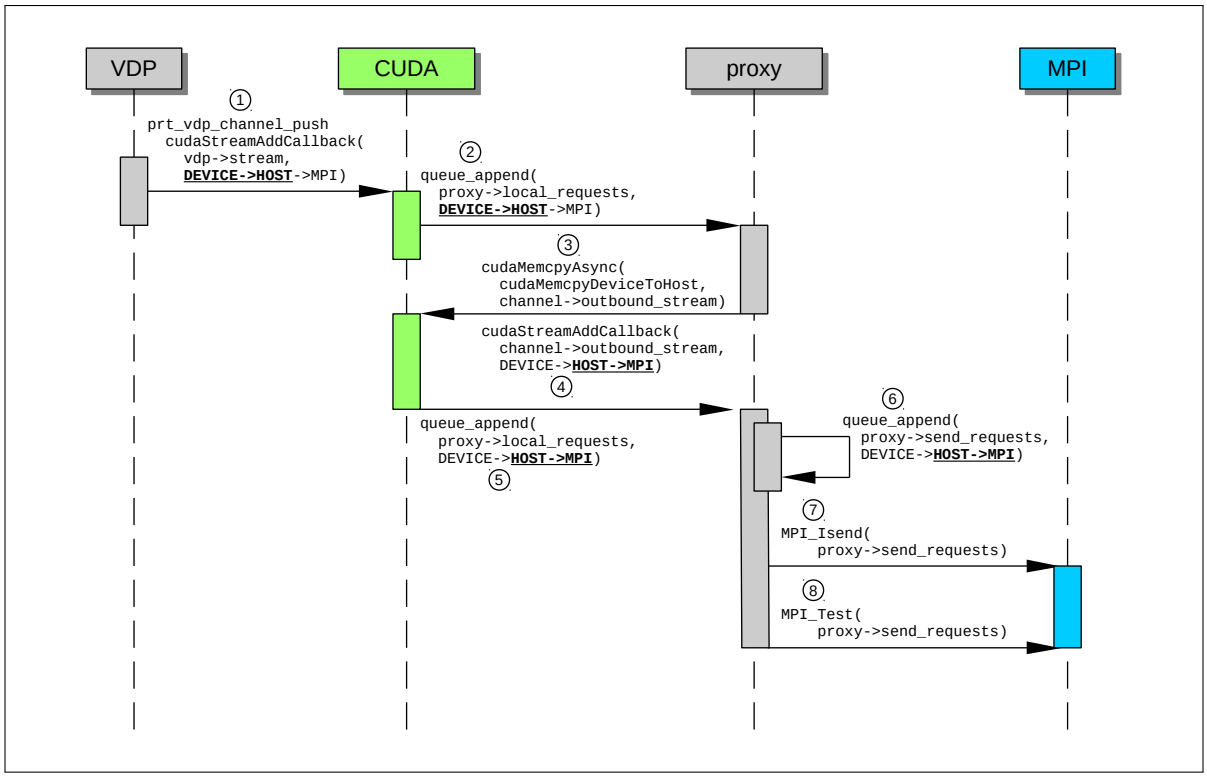


Figure 5. Timeline for a transfer originating from a GPU and involving MPI

the `-DMPI` flag in the compilation options and linking the application with an MPI library, in addition to the PULSAR library. The same goes for CUDA with the `-DCUDA` flag. At the same time, the use of `#define` directives is avoided thorughout the code. Instead of conditional blocks of code handling MPI and CUDA, the code is written as if MPI and CUDA were present. If they are not, a set of empty MPI/CUDA stub functions is included.

PULSAR only depends on the most basic data structures: a non-thread-safe double-linked list, a thread-safe double-liked list (implemented by protecting the non-thread-safe one with Pthread spinlocks), and a hash table. The basic linked list and hash table are implemented themselves as dependency-free, stand-alone structures.

PULSAR contains its own tracing routines, based on recording time of different events and writing an SVG file at the end of execution. PULSAR records computational tasks on CPUs and GPUs (VDP firings), MPI communications, and CUDA data transfers. CPU timestamps are taken using `get_time_of_day`, GPU timestamps are take using `cudaEventRecord`.

PULSAR code, including the runtime (PRT) and examples, is available on the project website (`https://bitbucket.org/icl/pulsar`), which also contains extensive documentation, including: installation instructions, users' guide, reference manual, etc. The code is documented using Doxygen and the reference manual is produced automatically from Doxygen annotations. A simple version is available in HTML and an extended version in PDF, where function call graphs and data structures dependency graphs are also included.

5. Cannon's Matrix Multiplication

Cannon's algorithm for matrix multiplication is arguably the best known systolic algorithm. Here it makes for a perfect example due to its simplicity and compactness of implementation. Figure 6 shows the basic principle of Cannon's algorithm. A two-dimensional (2D) mesh of

processors is used to compute the matrix product, $C = A \times B$, by rotating the A matrix from left to right, and the B matrix from top to bottom, while each processor computes a block of C.

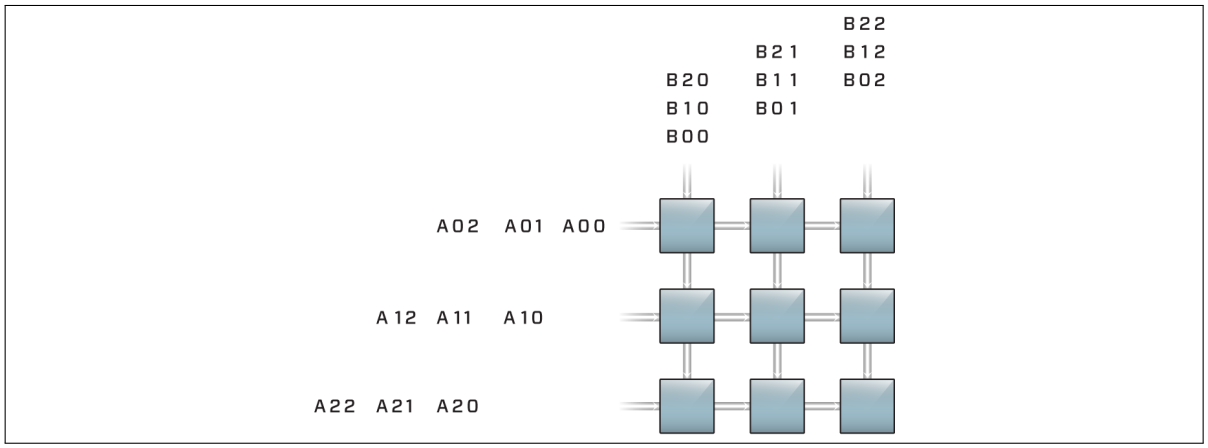


Figure 6. Cannon's matrix multiplication algorithm

Figure 7 shows the PULSAR code for the construction of the VSA. The basic premise of the implementation is to built a 2D mesh of VDPs, and let each VDP compute one tile of the result matrix C. Here nb is the size of each tile, and nt is the width and height of the matrix (number of tiles). The code loops over the vertical and horizontal dimensions of the matrix and, for each tile, creates a VDP, creates four channels (two vertical, two horizontal), inserts the channels in the VDP, and inserts the VDP in the VSA.

Each VDP holds its local tile of the C matrix, as well as its local tiles of matrices A and B, which are distributed in a skewed fashion, as depicted on Figure 6, i.e., tiles of the B matrix are shifted by one vertically, from column to column, and tiles of A are shifted by one horizontally, from row to row. All channels are initially inserted as inactive, so that each VDP can be launched without any data in the input channels, compute the local part of the product, send its local tile of A to the right, and send its local tile of B down.

Figure 8 shows a complete code of a VDP implementing the Cannon's algorithm. It starts with the declarations section, where tile size (nb) and matrix size (nt) are retrieved from `global_store`, the location of the VDP tile in the matrix (m, n) is retrieved from the VDP tuple, an alias is created to the `local_store`. The `done` variable is declared because the cuBLAS calls require passing of the constant by reference.

The first half of the code handles the first step, when the VDP's counter equals to the size of the matrix. If the VDP is a device VDP (is assigned to a GPU), then cuBLAS initializations are handled in the first step (creating cuBLAS handle and associating it with the VDP's stream). Then the local tile A is pushed to the right and the local tile of B is pushed down. Then the local product is computed, either using a cuBLAS call or a CBLAS call, depending on the VDP's location. Finally, the input channels are activated to provide data for the next step.

In the followup steps, a tile of A is read from the left and passed to the right, unless it is the last step of the algorithm (VDP reaches one). Tiles of B are handeld in the same manner, passing downwards. Then the product is computed using an appropriate call (either cuBLAS or CBLAS), and finally the transient packets, used for transfers of A and B, are released.

```
#define LEFT_INPUT  0
#define RIGHT_OUTPUT 0
#define UPPER_INPUT  1
#define LOWER_OUTPUT 1
void vsa_init(prt_vsa_t *vsa, int nb, int nt) {
    int m;
    int n;
    for (m = 0; m < nt; m++) {
        for (n = 0; n < nt; n++) {

            prt_vdp_t *vdp = prt_vdp_new(
                prt_tuple_new2(m, n), nt,
                vdp_func_dgemm, sizeof(local_store_t), 2, 2);

            prt_channel_t *channel;
            channel = prt_channel_new(
                nb*nb*sizeof(double),
                prt_tuple_new2(m, n > 0 ? n-1 : nt-1), RIGHT_OUTPUT,
                prt_tuple_new2(m, n), LEFT_INPUT);
            prt_channel_off(channel);
            prt_vdp_channel_insert(vdp, channel, PRT_INPUT_CHANNEL);

            channel = prt_channel_new(
                nb*nb*sizeof(double),
                prt_tuple_new2(m, n), RIGHT_OUTPUT,
                prt_tuple_new2(m, n+1 < nt ? n+1 : 0), LEFT_INPUT);
            prt_vdp_channel_insert(vdp, channel, PRT_OUTPUT_CHANNEL);

            channel = prt_channel_new(
                nb*nb*sizeof(double),
                prt_tuple_new2(m > 0 ? m-1 : nt-1, n), LOWER_OUTPUT,
                prt_tuple_new2(m, n), UPPER_INPUT);
            prt_channel_off(channel);
            prt_vdp_channel_insert(vdp, channel, PRT_INPUT_CHANNEL);

            channel = prt_channel_new(
                nb*nb*sizeof(double),
                prt_tuple_new2(m, n), LOWER_OUTPUT,
                prt_tuple_new2(m+1 < nt ? m+1 : 0, n), UPPER_INPUT);
            prt_vdp_channel_insert(vdp, channel, PRT_OUTPUT_CHANNEL);

            prt_vsa_vdp_insert(vsa, vdp);
        }
    }
}
```

Figure 7. VSA implementing the Cannon's algorithm

6. Performance Experiments

PULSAR's capabilities are demonstrated using a simple weak scaling experiment carried out on up to 1024 nodes of the Titan supercomputer. Each node contains a 16-core AMD Interlagos CPU and one NVIDIA Tesla K20X GPU. In this experiment, the nodes are assigned in a square array of size 2×2, 4×4, 8×8, 16×16 and 32×32. Each node is assigned a 4×4 array of tiles, where each tile is of size 2048×2048. Runs are made using CPUs only or GPUs only. Multithreaded BLAS is used for the CPU runs and cuBLAS is used for the GPU runs.

Figure 9 shows the scaling. The dashed lines show ideal scaling, taking the smallest parallel case (2×2 nodes) as the reference point. Figure 10 shows an execution trace of a small CPU run (2×2 nodes, 2×2 tiles per node, 2048×2048 tiles). Figure 11 shows an execution trace of the same run using GPUs. In the CPU trace, the timeline of each node is represented by two lines. The first one shows the execution of the matrix multiplications. The second one shows invocations of communication tasks. In the case of communication tasks, only the duration of asynchronous MPI calls is registered (not duration of the actual communication), which results

```
void vdp_func_dgemm(prt_vdp_t *vdp) {
    int nb = ((global_store_t*)vdp->global_store)->nb;
    int nt = ((global_store_t*)vdp->global_store)->nt;
    int m = vdp->tuple[0];
    int n = vdp->tuple[1];
    local_store_t *local_store = (local_store_t*)vdp->local_store;
    double done = 1.0;

    // IF first step.
    if (vdp->counter == nt) {
        // IF device VDP.
        if (vdp->location == PRT_LOCATION_DEVICE) {
            cublasStatus_t cublas_status;
            cublas_status = cublasCreate(&local_store->cublas_handle);
            cublas_status = cublasSetStream(local_store->cublas_handle, vdp->stream);
        }
        // Push local A right.
        prt_packet_t *right_packet = prt_vdp_packet_new(vdp, nb*nb*sizeof(double), local_A);
        prt_vdp_channel_push(vdp, RIGHT_OUTPUT, right_packet);

        // Push local B down.
        prt_packet_t *lower_packet = prt_vdp_packet_new(vdp, nb*nb*sizeof(double), local_B);
        prt_vdp_channel_push(vdp, LOWER_OUTPUT, lower_packet);

        // Compute local AxB.
        if (vdp->location == PRT_LOCATION_DEVICE) {
            cublasDgemm(local_store->cublas_handle, CUBLAS_OP_N, CUBLAS_OP_N, nb, nb, nb,
                        &done, local_A, nb, local_B, nb, &done, local_C, nb);
        } else {
            cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, nb, nb, nb,
                        1.0, local_A, nb, local_B, nb, 1.0, local_C, nb);
        }
        // Activate input channels.
        prt_vdp_channel_on(vdp, LEFT_INPUT);
        prt_vdp_channel_on(vdp, UPPER_INPUT);
    } else {
        // Move A horizontally.
        prt_packet_t *left_packet  = prt_vdp_channel_pop(vdp, LEFT_INPUT);
        if (vdp->counter > 1)
            prt_vdp_channel_push(vdp, RIGHT_OUTPUT, left_packet);

        // Move B vertically.
        prt_packet_t *upper_packet = prt_vdp_channel_pop(vdp, UPPER_INPUT);
        if (vdp->counter > 1)
            prt_vdp_channel_push(vdp, LOWER_OUTPUT, upper_packet);

        // Compute AxB.
        if (vdp->location == PRT_LOCATION_DEVICE) {
            cublasDgemm(local_store->cublas_handle, CUBLAS_OP_N, CUBLAS_OP_N, nb, nb, nb,
                        &done, left_packet->data, nb, upper_packet->data, nb,
                        &done, local_C, nb);
        } else {
            cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, nb, nb, nb,
                        1.0, left_packet->data, nb, upper_packet->data, nb,
                        1.0, local_C, nb);
        }
        // Release transient packets.
        prt_vdp_packet_release(vdp, left_packet);
        prt_vdp_packet_release(vdp, upper_packet);
    }
}
```

Figure 8. VDP code implementing the Cannon's algorithm

in thin stripes in the trace. In the GPU trace, the timeline of each node is represented by three lines. The first one shows the execution of the matrix multiplications. The second one shows the DMA transfers between the host memory (CPU) and the device memory (GPU). Here, actual durations of the transfers are shows. The third one shows durations of the MPI calls, just like in the CPU case.

In the CPU case, MPI communication is completely overlapped with computation, resulting in no gaps in the trace and almost idea scaling. In the GPU case, the DMA transfers do not keep up with the speed of execution, and the MPI transfers do not keep up with the DMA transfers, resulting in large gaps in the trace and poor scaling. At the same time, GPU execution still produces much higher overall performance than CPU execution.

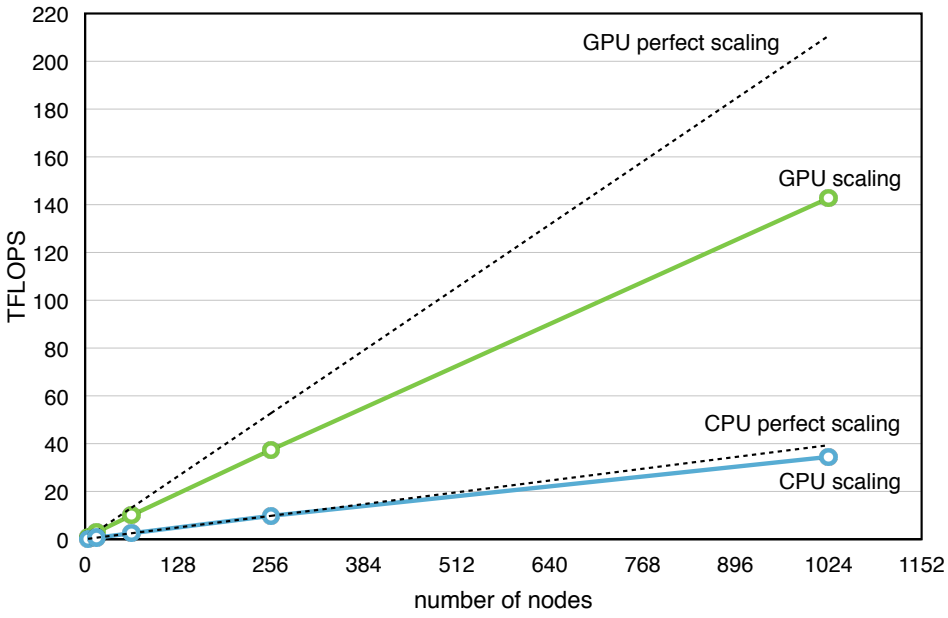


Figure 9. Scaling of Cannon's algorithm

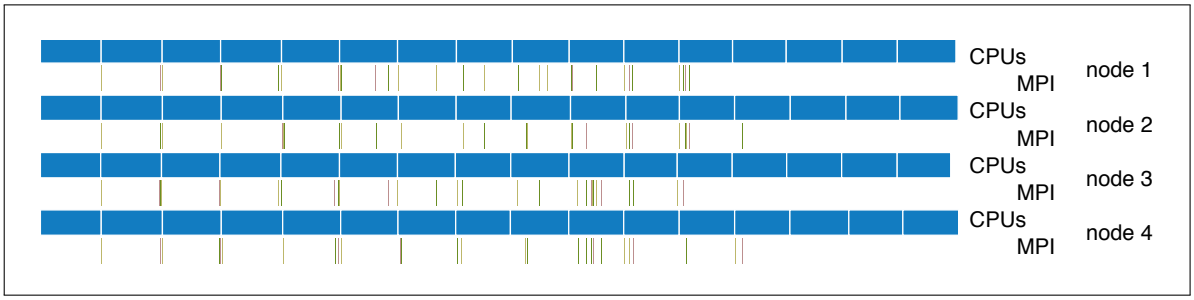


Figure 10. CPU trace using 4 nodes

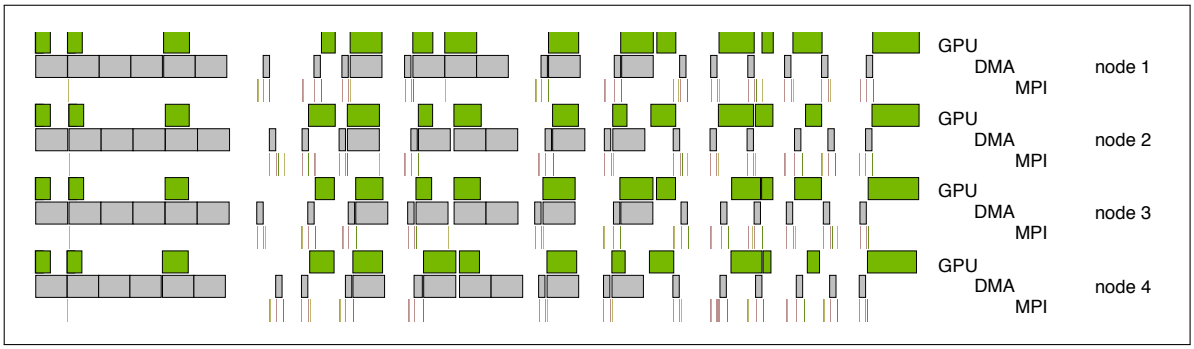


Figure 11. GPU trace using 4 nodes

Conclusion

This paper has presented the PULSAR system. PULSAR combines the key paradigms of systolic arrays (regularity, extensive data re-use and multilevel pipelining) with virtualization techniques, in order to provide a simple yet efficient programming model to design parallel algorithms on complex multicore systems with attached accelerators.

After detailing the nuts and bolts of the system, we have provided a comprehensive description of a PULSAR instance, namely the canonic Cannon's algorithm for matrix product. We have shown convincing performance results on Titan, which nicely demonstrate that a limited programming effort, as required by PULSAR, is not incompatible with an efficient implementation. Achieving a good trade-off between the ease of programming and the quality of the results was the primary objective of PULSAR.

Acknowledgements

This work has been supported by the National Science Foundation, under grant SHF-1117062, Parallel Unified Linear algebra with Systolic ARrays (PULSAR). The authors would also like to thank the National Institute for Computational Sciences, the Georgia Institute of Technology and the Oak Ridge National Laboratory for generous computer allocations on their supercomputers. Yves Robert has been supported by Institut Universitaire de France.

References

1. Ahmed, H.M., Delosme, J.M., Morf, M.: Highly concurrent computing structures for matrix arithmetic and signal processing. Computer 15(1), 65–82 (1982), DOI: 10.1109/MC.1982.1653828

2. Allen, E., Culpepper, R., Nielsen, J., Rafkind, J., Ryu, S.: Growing a syntax. In: ACM SIGPLAN Foundations of Object-Oriented Languages workshop. ACM, Savannah, GA, USA (2009)

3. Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Webb, J.A.: The Warp computer: Architecture, implementation, and performance. IEEE Transactions on Computers C-36(12), 1523–1538 (1987), DOI: 10.1109/TC.1987.5009502

4. Arpaci, R., Culler, D., Krishnamurthy, A., Steinberg, S., Yelick, K.: Empirical evaluation of the CRAY-T3D: A compiler perspective. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture. pp. 320–331. IEEE, Santa Margherita Ligure, Italy (June 22-24 1995), iSSN: 1063-6897, Print ISBN: 0-89791-698-0, INSPEC Accession Number: 5086797

5. Augonnet, C.: Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective. Phd thesis, Universit'e Bordeaux 1 (December 2011)

6. Barada, H., El-Amawy, A.: Systolic architecture for matrix triangularisation with partial pivoting. IEE Proceedings E, Computers and Digital Techniques 135(4), 208–213 (1987), ISSN: 0143-7062

7. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R.A.: The ILLIAC IV computer. IEEE Transactions on Computers C-17(8), 746–757 (1968), DOI: 10.1109/TC.1968.229158

8. Bojanczyk, A.W., Brent, R.P., Kung, H.T.: Numerically stable solution of dense systems of linear equations using mesh-connected processors. SIAM J. Sci. Stat. Comput. 5(1), 95–104 (1984), DOI: 10.1137/0905007

9. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, H., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., Dongarra, J.: Distibuted Dense Numerical Linear Algebra Algorithms on Massively Parallel Architectures: DPLASMA. Tech. rep., Innovative Computing Laboratory, University of Tennessee (apr 2010), `http://icl.cs.utk.edu/news_pub/submissions/ut-cs-10-660.pdf`

10. Cannon, L.E.: A Cellular Computer to Implement the Kalman Filter Algorithm. Ph.D. thesis, Montana State University (1969)

11. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Tech. Rep. CCS-TR-99-157, CCS (May 13 1999)

12. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. International Journal of High Performance Computing Applications 21(3), 291–312 (August 2007)

13. Chapel language specification 0.750 (2007), `http://chapel.cs.washington.edu/spec-0.750.pdf`, accessed: 2017-02-15

14. Chapman, B., Curtis, T., Pophale, S., Koelbel, C., Kuehn, J., Poole, S., Smith, L.: Introducing OpenSHMEM, SHMEM for the PGAS community. In: PGAS'10: The Fourth Conference on Partitioned Global Address Space Programming Models. PGAS, ACM, New York, NY, USA (October 12-15 2010)

15. Choi, J.: A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In: High Performance Computing on the Information Superhighway, 1997. HPC Asia '97. pp. 224–229. IEEE, Seoul (April-May 1997), DOI: 10.1109/HPC.1997.592151

16. Choi, J., Walker, D.W., Dongarra, J.J.: PUMMA: parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Concurrency Computat.: Pract. Exper. 6(7), 543–570 (1994), DOI: 10.1002/cpe.4330060702

17. Comon, P., Robert, Y.: A systolic array for computing BA^{-1}. IEEE Transactions on Acoustics, Speech and Signal Processing 35(6), 717–723 (1987), DOI: 10.1109/TASSP.1987.1165208

18. Consortium, U.: Upc language specifications, v1.2. Tech. Rep. LBNL-59208, Lawrence Berkeley National Laboratory (2005)

19. Darcy, J.: Writing robust IEEE recommended functions in "100% pure Java"(tm). Tech. Rep. CSD-98-1009, Computer Science Division, University of California, Berkeley (Oct 1998)

20. Dennis, J.B., Gao, G.R.: On the feasibility of a codelet based multi-core operating system. In: 4th Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM'14). Edmonton, Alberta, Canada (August 24 2014)

21. Dongarra, J., Graybill, R., Harrod, W., Lucas, R., Lusk, E., Luszczek, P., McMahon, J., Snavely, A., Vetter, J., Yelick, K., Alam, S., Campbell, R., Carringon, L., Chen, T.Y., Khalili, O., Meredith, J., Tikir, M.: Darpa's hpcs program: History, models, tools, languages. Advances in Computers: High Performance Computing 72, 1–100 (2008), iSBN: 978-0-12-374411-1, ISSN: 0065-2458

22. Evans, D.J.: Systolic Algorithms (Topics in Computer Mathematics). Routledge (1991), ISBN: 2881248047

23. Fanfarillo, A., Burnus, T., Cardellini, V., Filippone, S., Nagle, D., Rouson, D.W.I.: Opencoarrays: Open-source transport layers supporting Coarray Fortran compilers. In: 8th International Conference on Partitioned Global Address Space Programming Models. ACM, Eugene, Oregon, USA (October 7-10 2014)

24. Fisher, A.L., Kung, H.T.: Special-purpose VLSI architectures: General discussions and a case study. In: Kung, S.Y., Kailath, T., Whitehouse, H.J. (eds.) VLSI and Modern Signal Processing, pp. 153–169. Prentice Hall (1984), ISBN: 013942699X

25. Fortes, J.A.B., Wah, B.W.: Systolic arrays-from concept to implementation. Computer 20(7), 12–17 (1987), DOI: 10.1109/MC.1987.1663616

26. Fox, G.C., Otto, S.W., Hey, A.J.G.: Matrix algorithms on a Hypercube I: Matrix multiplication. Parallel Comput. 4(1), 17–31 (1987), DOI: 10.1016/0167-8191(87)90060-3

27. Gao, G.R., Sterling, T., Stevens, R., Hereld, M., Zhu, W.: Parallex: A study of a new parallel computation model (2007)

28. Gentleman, W.M., Kung, H.T.: Matrix triangularization by systolic arrays. In: SPIE Proceedings Vol. 298, Advances in Laser Scanning Technology. pp. 19–26. Society for Photo-Optical Instrumentation Engineers, Bellingham, WA (1981)

29. Gregory, J., McReynolds, R.: The SOLOMON computer. IEEE Transactions on Electronic Computers EC-12(6), 774–781 (1963), DOI: 10.1109/PGEC.1963.263560

30. Huss-Lederman, S., Jacobson, E.M., Tsao, A., Zhang, G.: Matrix multiplication on the Intel Touchstone Delta. Concurrency: Pract. Exper. 6(7), 571–594 (1994), DOI: 10.1002/cpe.4330060703

31. Jin, G., Adhianto, L., Mellor-Crummey, J., III, W.N.S., Yang, C.: Implementation and performance evaluation of the hpc challenge benchmarks in coarray Fortran 2.0. In: 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS). Anchorage, AK, USA (May 16-20 2011)

32. Johnson, K.T., Hurson, A.R., Shirazi, B.: General-purpose systolic arrays. Computer 23(11), 20–31 (1993), DOI: 10.1109/2.241423

33. Jr., G.L.S., Allen, E., Chase, D., Flood, C., Luchangco, V., Maessen, J.W., Ryu, S.: Fortress (Sun HPCS language). In: Encyclopedia of Parallel Computing, pp. 718–735. Springer, New York Dordrecht Heidelberg London (2011), DOI 10.1007/978-0-387-09766-4

34. Kaiser, H., Brodowicz, M., Sterling, T.: Parallex: An advanced parallel execution model for scaling-impaired applications (2009)

35. Kaiser, H., Brodowicz, M., Sterling, T.: Parallex. 2012 41st International Conference on Parallel Processing Workshops 0, 394–401 (2009)

36. Kale, L.V., Krishnan, S.: CHARM++: A portable concurrent object oriented system based on C++. SIGPLAN Not. (10), 91–108 (Oct.), DOI: 10.1145/167962.165874

37. Krishnamurthy, A., Culler, D., Yelick, K.: Evaluation of architectural support for global address-based communication in large-scale parallel machines. Tech. Rep. CSD-98-984, Computer Science Division, University of California, Berkeley (1998)

38. Krishnamurthy, A., Yelick, K.: Optimizing parallel programs with explicit synchronization. In: Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation. pp. 196–204. ACM (Jun 1995)

39. Krishnamurthy, A., Yelick, K.: Analyses and optimizations for shared address space programs. Journal of Parallel and Distributed Computation 38(2), 130–144 (November 1 1996)

40. Kuck, D.J.: ILLIAC IV software and application programming. IEEE Transactions on Computers C-17(8), 758–770 (1968), DOI: 10.1109/TC.1968.229159

41. Kung, H.T.: Why systolic architectures? Computer 15(1), 37–46 (1982), DOI: 10.1109/MC.1982.1653825

42. Kung, H.T., Leiserson, C.E.: Systolic arrays (for VLSI). In: Sparse Matrix Proceedings. pp. 256–282. Society for Industrial and Applied Mathematics (1978), ISBN: 0898711606

43. Kung, S.Y., Lo, S.C., Jean, S.N., Hwang, J.N.: Wavefront array processors-concept to implementation. Computer 20(7), 18–33 (1987), DOI: 10.1109/MC.1987.1663617

44. Liblit, B.: Local Qualification Inference for Titanium. `http://www.cs.berkeley.edu/~liblit/lqi/` (Aug 26 1998), cS263/CS265 semester project report.

45. Liblit, B., Aiken, A.: Type systems for distributed data structures. In: In the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 199–213. Boston, Massachusetts (19–21 Jan 2000)

46. Luk, F.T.: Triangular processor array for computing singular values. Lin. Alg. Appl. 77, 259–273 (1986), DOI: 10.1016/0024-3795(86)90171-0

47. Marjanović, V., Labarta, J., Ayguadé, E., Valero, M.: Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In: Proceedings of the 24th ACM International Conference on Supercomputing. pp. 5–16. ICS '10, ACM, New York, NY, USA (2010), DOI: 10.1145/1810085.1810091

48. Mellor-Crummey, J., Adhianto, L., Jin, G., III, W.N.S.: A new vision for coarray fortran. In: The Third Conference on Partitioned Global Address Space Programming Models. Ashburn, VA, USA (October 5-8 2009)

49. Mellor-Crummey, J., Adhianto, L., Scherer, W.N.: A critique of co-array features in fortran 2008 working draft j3/07-007r3 (February 2008), paper J3 08-126 of the Fortran 2008 J3 standard working group

50. Miyamoto, C., Liblit, B.: Themis: Enforcing Titanium Consistency on the NOW. `http://www.cs.berkeley.edu/~liblit/themis/` (Dec 1997), cS262 semester project report.

51. Numrich, R.W., Reid, J.K.: Co-array Fortran for parallel programming. ACM SIGPLAN Fortran Forum 17(2), 1–31 (Aug 1998), DOI: 10.1145/289918.289920

52. Quinton, P., Robert, Y.: Systolic Algorithms & Architectures. Prentice Hall (1991), ISBN: 0138807906

53. Robert, Y.: Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm. Manchester University Press (1991), ISBN: 0470217030

54. Scherer, W.N., Adhianto, L., Jin, G., Mellor-Crummey, J., Yang, C.: Hiding latency in Coarray Fortran 2.0. In: PGAS'10: The Fourth Conference on Partitioned Global Address Space Programming Models. PGAS, New York, NY, USA (October 12-15), DOI: 10.1145/2020373.2020387

55. Shterenlikht, A., Margetts, L., Cebamanos, L., Henty, D.: Fortran 2008 CoArrays. ACM SIGPLAN Fortran Forum 34(1), 10–30 (Apr 2015), DOI:10.1145/2754942.2754944

56. Sorensen, D.C.: Analysis of pairwise pivoting in Gaussian elimination. IEEE Transactions on Computers C-34(3), 274–278 (1985), DOI: 10.1109/TC.1985.1676570

57. Tan, A.T., Falcou, J., Etiemble, D., Kaiser, H.: Automatic task-based code generation for high performance domain specific embedded language. International Journal of Parallel Programming (2015)

58. Van De Geijn, R.A., Watts, J.: SUMMA: scalable universal matrix multiplication algorithm. Concurrency Computat.: Pract. Exper. 9(4), 255–274 (1997), DOI: 10.1002/(SICI)1096-9128(199704)9:43.0.CO;2-2

59. Report on the experimental language X10, version 1.0.1 (December 2006), `http://x10.sourceforge.net/docs/x10-101.pdf`, accessed: 2017-02-15

DOI: 10.14529/jsfi170102

Workflows for Science: a Challenge when Facing the Convergence of HPC and Big Data

Rosa M. Badia[1,2]***, Eduard Ayguade***[1,3]***, Jesus Labarta***[1,3]

Workflows have been traditionally a mean to describe and implement the computing experiments, usually parametric studies and explorations searching for the best solution, that scientific researchers want to perform. A workflow is not only the computing application, but a way of documenting a process. Science workflows may be of very different nature depending on the area of research, matching the actual experiment that the scientist want to perform. Workflow Management Systems are environments that offer the researchers tools to define, publish, execute and document their workflows.

In some cases, the science workflows are used to generate data; in other cases are used to analyse existing data; only in a few cases, workflows are used both to generate and analyse data. The design of experiments is in some cases generated blindly, without a clear idea of which points are relevant to be computed/simulated, ending up with huge amount of computation that is performed following a brute-force strategy.

However, the evolution of systems and the large amount of data generated by the applications require an in-situ analysis of the data, thus requiring new solutions to develop workflows that includes both the simulation/computational part and the analytic part. What is more, the fact that both components, computation and analytics, can be run together will enable the possibility of defining more dynamic workflows, with new computations being decided by the analytics in a more efficient way.

The first part of the paper will review current approaches that a set of scientific communities follow in the development of their workflows. This paper does not intent to be exhaustive in the compilation of different approaches available to develop and deploy workflows. We focus on the Workflow Management Systems used by a set of scientific communities and their representative use cases, with the objective of understanding their different needs and requirements. The second part of the paper proposes a new software architecture to develop a new family of end-to-end workflows that enables the management of dynamic workflows composed of simulations, analytics and visualization, including inputs/outputs from streams.

Keywords: workflows, scientific applications, Big Data.

Introduction

Workflows appeared last century and have been used in the manufacturing industry as a mean to optimize their processes. Examples of traditional (non-IT) workflows can be found in the assembly lines, i.e., the Ford Model T assembly line standardized the production processes and was the first continuous delivery pipeline for the automotive industry. This process reduced the costs of manufacturing from $850 to $260 in 1924.

The time and motion studies defined by Taylor [52] and Gilbreth [41] had significant impact in the manufacturing processes. These studies proposed to break manufacturing activities into small, simple steps, to determine with accuracy the amount of time required to perform each of the steps. Then, the sequence of movements taken by the employee has to be carefully observed to detect and eliminate redundant or wasteful motion, and the precise time invested for each correct movement is measured. From these measurements, production and delivery times and

[1]Barcelona Supercomputing Center (BSC), Barcelona, Spain
[2] Consejo Superior de Investigaciones Científicas (CSIC), Madrid, Spain
[3] Universitat Politècnica de Catalunya (UPC), Barcelona Spain

prices can be computed and incentive schemes devised. Methods used in these early times were: Flow diagrams, Gantt charts, and ERT charts.

Although the term workflow was not used at that time, the same concept is used in current Workflow Management Systems, a software system that is able to orchestrate a set of tasks. The tasks show dependencies between them, which can be of data or control, forming a task graph or workflow. The concept of workflow is used extensively in a large number of scientific communities.

Scientific users have a plethora of Workflow Management Systems available for their needs. Traditionally different communities stick to a system or to a set of systems for different reasons: due to the needs of the community applications, due to the popularity of given systems, due to historical reasons, to availability of domestic systems that are adopted by others and later extended, due to the possibility of sharing, avilability of specific functionalities that are needed by the community applications not present in others, etc. However, we believe that aspects such as modularity and elegance of the design, portability, genericity of the systems; should be given more attention.

The paper takes into account a set of Workflow Management Systems used by given scientific communities to implement their workflows: life science (genomics), earth-science (climate), fusion, and astrophysics. For each of them an example of how the workflows are defined and the specific features they have is described.

It is very usual that these scientific applications generate a large amount of data, and this is in-crescendo. Also, the use of parallel systems and High Performance Computing (HPC) is every time more usual. Traditionally, the phases of computation/simulation of these workflows have been decoupled from the phases of data analysis. Also, traditionally workflows are defined quite statically, even loops are possible, but no margin for dynamicity on the decision of what computations should be performed is left.

Taking into account potential users of next coming exascale architectures, workflow management systems that support the convergence of the computation and data analysis parts are a must. Even more, those workflows should support in-situ data-analysis and dynamism, in such a way that results from previous analysis determine the next steps of the workflow, i.e., which computation to trigger, searching for new alternatives or going in-depth into a more detailed simulation.

In section 1 we give an overview of the alternatives in the implementation of a Workflow Management System. Then, the paper is organized around the different cases that have been chosen: section 2 describes Kepler, and its usage by the fusion community; section 3 describes Pegasus, and the case of the LIGO collaboration that has been using this system for more than 10 years; section 4 describes Galaxy and its use in the framework of the Life Sciences community; section 5 describes the workflow management systems used by the Earth Science (climate) community; and section 6 describes Taverna and its use by the astrophysics community. Section 7 proposes a new architecture of end-to-end workflows with dynamic management, orchestrating the computation and analytics of the experiments. Final section concludes the paper.

1. Workflow Management Systems: an Overview

A Workflow Management System can be defined as a software environment able to orchestrate the execution of a set of interdependent computing tasks that exchange data between them with the objective of solving a given experiment. A workflow can be graphically described as

a graph, where the nodes denote the computations and the edges data or control dependencies between them.

Workflow Management Systems became very popular with the appearance of Grid computing, since they offered the possibility of exploiting this distributed infrastructure. Papers [14, 57] present taxonomies of Workflow Management Systems from that period. Some of the systems developed at that time are still alive projects used in current distributed computing platforms (either High Performance Computing (HPC) clusters, High Throughput Computing (HTC) platforms, clouds or combination of several of these options).

Workflows can be described graphically, with a drag and drop interface where the workflow is totally specified with a graphical interface by the user like in Kepler [6],Taverna [26], or Galaxy [3]. It can be described textually, by specifying the graph in a textual mode, indicating the nodes and its interconnections like in Pegasus [15] or ASKALON [21]. It can also be described programmatically, using all the flexibility of a programming language to describe the behaviour of the workflow that is dynamically built depending on the actual dependencies found by the workflow system like in PyCOMPSs/COMPSs [32] or Swift [55]. A particular case of this is the use of simple tagged scripts that are processed by the actual engine, like with Cylc [39], Autosubmit [34], or ecFlow [33]. Another alternative is to describe the workflow through a set of commands with a command interface, like with Copernicus [42]. With the objective of offering a single syntax to describe workflows the initiative of the Common Workflow Language [7] has appeared. The Common Workflow Language (CWL) is a working group consisting of various organizations with interest in portability of data analysis workflows, mostly oriented to bioinformatics tools and with an emphasis on systems enabled with Docker. CWL offers a syntax to connect command line tools in order to create workflows that can be used by multiple platforms. CWL follows JASON or YAML syntaxes, or a mixture of the two.

Some systems orchestrate already deployed web services (Taverna), others compose external binaries or tools (Galaxy), and a few are able to interoperate directly with methods described in programming languages (PyCOMPSS/COMPSs). The data exchanged between the computation nodes of the workflows is typically a file, although in some cases can be objects in memory (like in PyCOMPSs/COMPSs).

A key component in a Workflow Management System is its engine. The engine is the responsible for coordinating the execution of all the tasks, scheduling them in the available computing resources and storage devices, transferring the data between distributed storage systems, monitoring the execution of the tasks, etc. The information that can be obtained about the engine in the literature is very variable: while for some systems (i.e. Pegasus, PyCOMPSs/COMPSs or Swift) the bibliography details sophisticated engines that implement various optimizations, either to schedule in parallel the workflow to be executed, to improve data locality, to be able to exploit heterogeneous computing platforms, ...; for others the information is very scarce and difficult to find.

On the user side, aspects that are valued by the scientific community are the possibility of sharing their workflows and data, and the support for workflow provenance. Several systems report the existence of repositories for workflows or experiments, like the myExperiment [23] repository, which currently supports inputs from several systems (Taverna, Galaxy and Kepler), or HUBzero [37] a software platform to support collaborations that is able to launch Pegasus workflows.

Another characteristic of these systems is the computing platform where the workflows are executed. As said before, many systems began their developments with the Grid as a computing platform, and still are able to run in this type of platforms, like the OSG [40] or EGI [30]. Most systems can execute in distributed environments (either composed of regular servers/clusters or HPC systems), also support for Clouds is common, and some systems are starting to support containers. While in most scientific communities the workflow tasks have been mostly sequential, the trend in general is to take benefit of current multicore architectures and accelerators such as GPGPUs or FPGAs, including tasks in the workflows that require some level of parallelism although with a low degree and only intranode (up to a few threads), other communities have been using large clusters or supercomputers for part of their workflow tasks (like in the climate or fusion communities). The trend in general is to take benefit of current multicore architectures, including tasks in the workflows that require some level of parallelism.

Given the amount of systems available, there have been some interoperability initiatives, like the European project SHIWA [49] and its continuation ER-flow [18] that dealt with interoperability of a dozen of workflow management systems existent at that time. The SHIWA simulation platform consists of a repository that supports the storage of workflows and metadata and of a portal that includes a workflow engine able to orchestrate workflows from different systems.

2. Kepler - Fusion community

The Kepler system [6] is free and open source, and developed, supported and maintained by the Kepler Project [29]. Kepler is a successor of the Ptolemy II system [43], and was designed to help users to create workflows, to perform analysis, to share and reuse workflow components, models and data between scientists. It was not designed to fulfill the needs of a specific community. With regard to data, Kepler is interoperable with a variety of formats, and supports local and remote data-access. The Kepler Project claims that the system is an effective environment to integrate software components of different nature, such as “R” scripts and compiled “C” code, or to facilitate the remote, distributed execution of models. This is done through the Java Native Interface and by using specific “Actors” (see below).

Kepler is based on a graphical user interface, where users select and connect the elements that will conform their scientific workflows, from computation, analysis and data sources.

Workflow components in Kepler are called Actors. Actors may contain a hierarchy of Actors, and in this case are called Composites. The Ports are the elements in the Actors that can receive Tokens. Tokens may include single or multiple data or messages. The execution of workflows is controlled by Directors in Kepler. Typically, a Director manages the execution of a set of actors. Actors can be tuned with Parameters. Kepler was extended to be able to access streaming sensor data and archived historical data [8]. In fig. 1 we can see a sample Kepler workflow that accesses sensor data.

Kepler actors are executed as local Java threads, but can also spawn distributed execution threads via web services or through the Java Native Interface (JNI). The actual execution model of the workflow depends on the nature of the director: for example, an SDF director will imply a synchronous execution of the workflow, where each computation node is processed one after the other; a PN Director will imply an execution of the workflow actors in parallel.

Kepler is a Java-based application that is maintained for the Windows, OSX, and Linux operating systems.

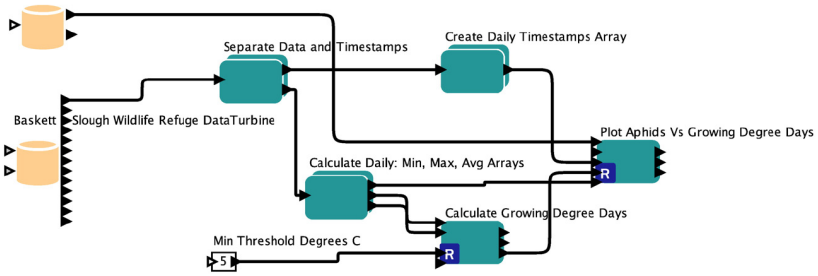


Figure 1. Sample Kepler analysis workflow which includes sensor data (taken from [8])

2.1. Use of Kepler in the Fusion Community

The fusion community in Europe is organized around EUROfusion, the European consortium for the development of Fusion Energy [20]. In the framework of the EUROfusion project, the European Integrated Modelling (EU-IM) team has as objective the development of a tokamak simulator that considers both the physics and all the machine related data, applicable to any fusion device. The simulation platform has been designed to be modular, flexible, and independent of a programming language. In 2011, the community evaluated different existing workflow engines and selected Kepler for the development of their workflows [27].

With this objective, they built a modelling infrastructure with a generic data structure that integrates both simulated and experimental data. The elements of this data structure are identified as "Consistent Physical Objects" (CPO). Thanks to this standardization of elements as CPOs, modules that solve the physics can be coupled into different integrated simulations (workflows). Also, modules describing the same physics can be interchanged within the same workflow. Physics modules are mapped as actors of a Kepler workflow and the data transfer among actors are performed through CPOs. Thanks to the semantic types that can be defined in Kepler, different CPOs can be distinguished and it can be verified if the different actors are correctly connected between them. Another feature interesting to this community is the functionality that Kepler allows for interactive steering of simulations, enabling to pause the simulation and reconfigure it, as well as the possibility of visualizing the present state of a simulation with specific actors.

The applications of the EU-IM require to execute from simple orchestration of workflows without convergence loops to tightly coupled workflows, involving mutual interactions among different codes.

An example of tightly coupled workflow has been built by the EU-IM [22] (formerly, EFDA ITM-TF). The European Transport Simulator (ETS) workflow [13], which couples different codes and will enable an entire discharge simulation from the start up until the current termination phase, including controllers and sub-systems. This workflow includes parallel components, like the GEMHPC one, which is run in 1024 cores. GEMHPC is based in GEM, which is written in MPI [48].

Within the project EUFORIA, the joint usage of different computing infrastructures (both HPC and HTC) in the Fusion community was considered. The solution derived by this project leverages and integrates different existing middleware: Kepler, as a workflow engine, which accesses the infrastructure using the Roaming Access Server (RAS). RAS provides access to the different underlying infrastructures using two alternative middlewares: gLite [31] and UNICORE [19]. Also, interactive access to the resources is supported with i2glogin [11].

The use of the different type of resources in this project took into account that, generally, simulations generating large amount of data will require large computing power only found in HPC systems, while the data analysis phase can be performed as independent tasks in HTC servers.

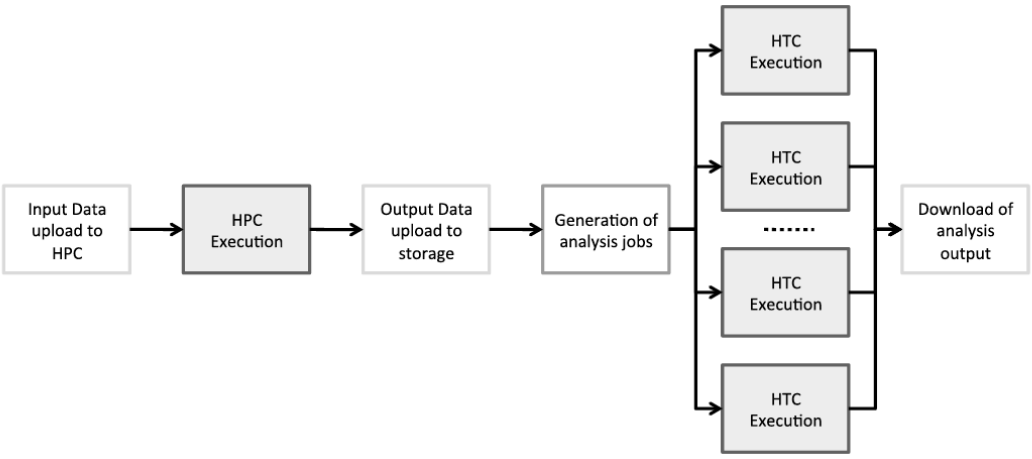


Figure 2. Sample Hybrid workflow from fusion community

The scenario considered in this project consisted of large simulations performed in HPC systems and the data produced was transferred to a storage system (see fig. 2). This transfer can be concurrent to the simulation, thus allowing to begin some of the following steps of the workflow.

3. Pegasus and the LIGO Collaboration

Pegasus [15] is a system based on the idea that users should define abstract workflows that include computations and information about the data without a direct mapping into the actual compute and storage resources. In that system, workflows are described as Direct Acyclic Graphs (DAGs), where nodes represent computational tasks and the edges represent data and control dependencies between the tasks. The data is exchanged between tasks in the form of files. From the abstract workflow, the actual physical location of data and executables is decided by the *Mapper*, which converts the abstract workflow into an executable one, with all the information about the location of the data files, resources where to execute the computations, etc. To locate the resources and files, catalogs are used that contain all the information.

Pegasus Mapper modifies the initial DAG before execution, therefore, statically. Nodes can be removed if there is data that is already available. Another optimization that Pegasus performs is task clustering, that merges a set of short duration nodes into a single one to reduce overhead. Some of these clustering strategies are guided by the users. The Mapper also associates jobs with workflow engines (again statically).

The workflow can be submitted to a local computing environment, a remote physical cluster or grid, or a virtual computing environment like the cloud.

In Pegasus, the workflow management system takes care of all the activities related to the execution of the workflow, from job and data management, monitoring and failure handling.

Pegasus provides textual interfaces in different programming languages, such as Python, Java and Perl, but what is described in these languages is the explicit abstract workflow, with

information about the nodes and their interconnections. This textual input, its translated into an XML description of the abstract workflow (DAX) which is then executed by the Pegasus engine. The Pegasus team advocates the use of textual workflows versus graphical workflows, since they consider that complex patters are easier to describe in this way.

Pegasus workflows can be defined in a hierarchical way, with nodes representing another workflow. This also helps to improve scalability of the system, since Pegasus needs to parse the whole XML file describing the DAX and for large cases this will not fit in memory. However, each DAX is managed by a different instance of the Pegasus engine. The hierarchy is used also in cases where the location of input files is unknown.

Pegasus has a set of execution engines with different features: single-core, which runs a single task at a time; non-shared file system, which stages in and out the files required by each computation; and Pegasus MPI Cluster (PMC) which is able to execute a DAG in Petascale systems by means of running them in an architecture based in a single master and several workers. PMC handles multi-core tasks but only within a node.

3.1. The LIGO Scientific Collaboration

Pegasus is a workflow environment that has been used for many different applications from various fields, from genomics, climate modeling, generation of sky mosaics, neuroscience, etc. The large collaborations where Pegasus has participated and has been key in their workflow developments are the LIGO Scientific Collaboration [1], the Southern California Earthquake Center (SCEC) [46] , and the National Virtual Observatory [38].

The Laser Interferometer Gravitational Wave Observatory (LIGO) is a network of gravitational-wave detectors, with observatories in Livingston, LA and Hanford, WA. The purpose of this collaboration is to prove the existence of the gravitational waves predicted by Einstein's General Theory of Relativity. To try to detect these waves, the scientists use kilometer-scale interferometric detectors.

The data generated by the instruments is distributed on the partners' sites, and then workflows are executed on the resources of their sites. Pegasus discovers the required data for each workflow and feeds the sites with them. One example of such an application is a workflow that searches for compact binary inspiral signals. The LIGO workflows are complex in the number of tasks (over 1.5 million jobs) and their dependencies, and the size of the datasets being analyzed (approximately 10 TB).

A characteristic of these workflows is that sometimes the granularity of the tasks is too small: in these cases, the workflows benefit of the feature of Pegasus that can cluster multiple tasks into one. Also, sometimes, some parts of the input data is recallibrated, requiring to recompute the workflow. However, recomputing the whole workflow is very expensive and what Pegasus offers is the possibility of registering data already being produced with the objective of not reproducing the part of the workflow that already generated it.

While LIGO workflows were initially (2002) deployed on the LIGO Data Grid, more recently have been extended to compute in the Open Science Grid and XSEDE. In September 2015 the LIGO collaboration detected gravitational waves. This detection was verified by processing roughly five terabytes of data by the LIGO workflows, generating many petabytes of exported data and executed in a distributed computing infrastructure composed of multiple HPC sites. The PyCBC search pipeline used for this validation is composed of hundreds of thousands tasks. Although some of the tasks are threaded (like calls to FFTW library), most of them are

sequential and short tasks. To reduce the overhead of small tasks in a large HPC cluster, the Pegasus MPI Cluster execution engine was used to submit sub-workflows as monolithic jobs.

4. Life-sciences Community - Galaxy

Galaxy [3] is a web-based platform initially designed for life sciences workflows. It offers a public service and a collaborative environment which enables to share, through internet, analysis tools, genomic data, tutorial demonstrations, persistent workspaces, and publication services, all available through internet in public repositories.

Through a web browser interface, Galaxy users can edit their workflows in a graphical editor where workflows are created by connecting tools. A Galaxy workflow is a reusable template analysis that a user can run repeatedly on different data; each time a workflow is run, the same tools with the same parameters are executed. Interoperability with different programming languages is done by invoking binaries: Galaxy supports any tool or piece of software for which a command line invocation can be constructed. Besides the graphical interface, the users can use BioBlend [51] a Python programmatic API to define their workflows in a textual form and supporting more complex formats difficult to deal in a graphical way.

Galaxy has a significant community of users and developers. Galaxy pages are the principal means to communicate research performed using Galaxy. Pages are interactive, web-based documents that users can create to describe a complete genomics experiment. This allows users to document and publish their experiments with computational outputs, allowing others to view the experiment with all the details and enable total reproducibility.

Galaxy enables the users to import datasets from many data warehouses. It relies on the concept of Object Store, a file interface that acts as a layer between Galaxy and user datasets. The Object Store supports distributed datasets and the application can exploit data locality and submit jobs to the resource closer to the data. Also, it automatically generates and maintains metadata about the different aspects of each analysis: input datasets, tools used, parameter values, and output datasets.

Users can import existing *histories*[4] and workflows, and rerun them. Also, they can modify or extend the analysis. Galaxy's public web server processes about 5,000 jobs per day and there is a large number of groups not affiliated with the Galaxy team that have been using the system to perform different types of genomic research and have published their results in prominent journals as Science or Nature. Besides the public server, a local instance can also be deployed in the user premises. Additional to the Galaxy server, Galaxy workflows can be executed in the cloud through the CloudMan platform [4].

One of the drawbacks reported by Galaxy users, is the challenge of installing a Galaxy instance [25]. This has been recently fixed by making available a Docker image.

Galaxy team is also collaborating in the definition of the Common Workflow Language support.

4.1. Galaxy Workflows in Life Science

Galaxy is very popular for Next-Gen Sequencing data analysis since it has available a large collection of tools for genomics and sequence analysis. Galaxy repositories [54] list on the order of thousand tools, most of them specific to genomics and sequence analysis that are used to

[4] A history is a series of analysis steps

compose the workflows. Most of these tools are sequential and parallelism is only exploited at very low levels (for example, up to 16 cores in Stampede [5]).

With regard the data used in these workflows, it can involve many datasets of variable size. For example, input data sets with 1 - 10 large files of 1 - 10 GB each, during the analysis other datasets are referenced (genome datasets of 1-40 GB) and although several intermediate datasets are produced, the final results require a relatively small amount of storage size (<100MB).

ELIXIR [17], the distributed infrastructure for life-science information partly funded by the European Commission within the Research Infrastructures programme of Horizon 2020, is an example of usage of Galaxy in this community. Due to the large interest of their scientific community in Galaxy, they established a Galaxy Working Group to evaluate the technical strategy for Galaxy within the context of ELIXIR. Between the activities performed by this group (meetings, surveys and discussions), they generated a report with recommendations on the use of Galaxy [12].

According to this report, Galaxy is used in that community for data intensive analysis from different domains: Genomics, Transcriptomics, Proteomics, Systems Biology, Metabolomics, but also metagenomics, imaging, small RNAs, etc. The popularity of Galaxy in that area is due to the possibility that offers to users with limited or no knowledge of command line to perform data-intensive analyses.

The users of Galaxy in this context can execute their workflows in the global Galaxy server or in local instances of Galaxy installed in the users institutions' or partner institutions. Most of the institutions reported in the survey the use of a compute cluster to host the Galaxy server (52.63%) but the amount of cores available for Galaxy jobs is surprisingly small (for most cases, from 10 - 49 cores, and only 9% of cases more than 100 cores).

BioExcel [10], the Center of Excellence for Computational Biomolecular Research funded by the European Commission, also has Galaxy as one of the workflow managers considered to be used in their activities. However, in this case, other systems have been considered due to the expertise of the partners: Taverna and PyCOMPSs/COMPSs, or the combination of two of them, for example, using Galaxy to compose coarse grain workflows and PyCOMPSs/COMPSs for a finer grain workflows that better exploit the parallelism of the system. Other workflow management systems considered by this community are KNIME [9] and Copernicus [42].

5. Cylc, Autosubmit and ecFlow and the Earth Science Community

The Earth Science community (climate) is another community case considered in this paper. Three different workflow management systems were compared by this community in the European project IS-ENES2 [28] which involved the stakeholders in Europe in that topic. The community considered: Cylc [39], Autosubmit [34] and ecFlow [33]. Although the community does not have a clear winner, the Met Office is using Cylc and received funding to continue development of Cylc.

Cylc is a Python based workflow engine and meta-scheduler. According to the developers, it specialises in continuous workflows of cycling tasks such as those used in weather and climate forecasting and research (i.e. workflows that show iterative patterns). Cylc is also easy to use with non-cycling systems. Cylc was created at the National Institute of Water and Atmospheric Research (NIWA, New Zealand) and is free software under the GNU GPL v3 license.

Cylc was developed to offer a workflow management system for the weather and climate community which based its studies on the use of complex scripts. Cylc is widely used by the community from research to real-time operations including ensemble prediction systems.

To provide robustness when executing the workflows, Cylc dumps state files and writes information into SQLite databases about the state of the execution. Cylc tasks can be configured to retry a number of times on failures.

Autosubmit is a solution created at IC3's Climate Forecasting Unit (CFU) to manage and run the research group's experiments. The development of this tool was a result of the lack of in house HPC facilities that led to a software design with very minimal requirements on the HPC that will run the jobs. Autosubmit, written in Python, provides a simple workflow definition and is capable to run experiments on remote clusters or supercomputers and on any GNU/Linux or Unix host. Autosubmit is currently being developed at the BSC Computational Earth Sciences group [35].

It has some fault tolerance features, based on check-pointing the tasks that have been finished: it keeps a list of completed tasks, and if the scheduler does not respond properly, when restarting the experiment the process will continue from the same point. A number of given retrials can also be defined for the different jobs that compose an experiment.

ecFlow is a workflow package that enables users to run a large number of programs (with dependencies on each other and on time) in a controlled environment. It is used at ECMWF to manage around half of their operational suites across a range of platforms. ecFlow checkpoint file allows it to restart at the last checkpoint before a failure. Also, a number of retries are supported on job failure.

The three systems have a similar input interface, based on scripts with tags, which seem to fulfill the needs of the community to describe their workflows. Cylc and ecFlow have also graphical interfaces to monitor the evolution of the execution of the experiments. While Cylc and ecFlow have a Graphical User Interface, Autosubmit only has some visual features through its monitor command.

5.1. Multi-member Climate Experiments with Autosubmit

The experiments that this community run are (multi-model) multi-member ensemble experiments. These experiments are traditionally organized in multiple simulations executed for given start dates (the purpose is to simulate weather or climate conditions on that period of time). The complexity of each experiment can be defined by different axes: number of start dates, number of members within a start date and number of chunks within a member.

For example, in [34] the authors present two experiments performed with Autosubmit. The experiments involved three type of resources: the local machine where the whole experiment is submitted, the MareNostrum3 supercomputer where the parallel simulations were run, and a post-processing fat node. In this case, each experiment consisted of 10 members of 4-month length for 34 start dates between 1993 and 2009 (only one chunk per member in this case). The total experiment consisted of 340 independent cases of 4 months, which is equivalent in cost to running a single simulation of approximately 113 years. This information is registered in an input configuration file which is provided as input to Autosubmit – this is how the user specify the workflow in this system.

Each simulation itself consists of several tasks: input data transfer, compilation, initialisation, chunk simulation, chunk post-processing, cleaning, and results data transfer. The user

needs to supply for each of these tasks the corresponding run scripts and the definition of how these tasks are to be executed (execution script, reference to computing resource to execute the task, dependencies with other tasks, number of processors required, etc).

At execution time, Autosubmit takes all the information from the configuration file and builds a task graph that takes into account the dependencies. Autosubmit is able to execute in parallel several tasks if dependencies between them allow it, although these type of experiments tend to be highly sequential since results from previous simulations are used as input for subsequent ones.

For each of these two experiments, Autosubmit ran 2381 jobs: 341 jobs were run in the local machine, 1360 in MareNostrum3 and 680 in the post-processing fat node. Some of the jobs in MareNostrum are MPI simulations using between 300-400 processors each.

6. Astrophysics - Taverna and COMPSs

Taverna [56] is another alternative Workflow Management System. Developed and maintained by the University of Manchester, it is currently used by several scientific communities.

Written in Java, it is composed of the Taverna Engine (used for enacting workflows), the Taverna Workbench (a graphical desktop client application, although a command line interface is also offered) and the Taverna Server (which supports the execution of remote workflows). Taverna supports local and remote services, and has been used in several domains: from biology, chemistry and medicine to music, meteorology and social sciences. The system is open source and it is offered for windows, linux and Mac OS.

Taverna [26] was initially designed as an application to ease the use of molecular biology tools and databases available on the web, especially web services. Taverna was designed with the philosophy that scientists could develop their workflows of webservices already published and then save the workflow in a repository, in such a way that the workflow can be reused and shared. The workflows are published in a public repository in http://www.myexperiment.org [23]. The myExperiment workflows repository does not only contain Taverna workflows, but also Galaxy or Kepler workflows.

New workflows are built with the Taverna Workbench in a graphical way, dragging and dropping new services in the workflow diagram and connecting their inputs and outputs. Taverna workflows are traditionally a mixture of web services, scripts (in R, for example) and other type of services.

In 2013 the Taverna engine was improved in order to be able to support scalable processing of large data sets, and to be capable of performing implicit iteration, looping and streaming of data. It was also at that time that the Taverna server was introduced, in order to support distributed execution.

The workflows can be executed on local machines or in distributed computing infrastructures (supercomputers, Grids or cloud environments), through the Taverna Server. An installation of the Server provides access to a collection of workflows (normally through a web interface, called the Taverna Player). However, in this execution mode users cannot edit the published workflows in the Server, neither add new workflows to the set of workflows deployed in the Server.

Another feature of Taverna is the possibility of tracking provenance: the Taverna engine records service invocations, intermediate and final workflow results. Also, Taverna supports nested workflows.

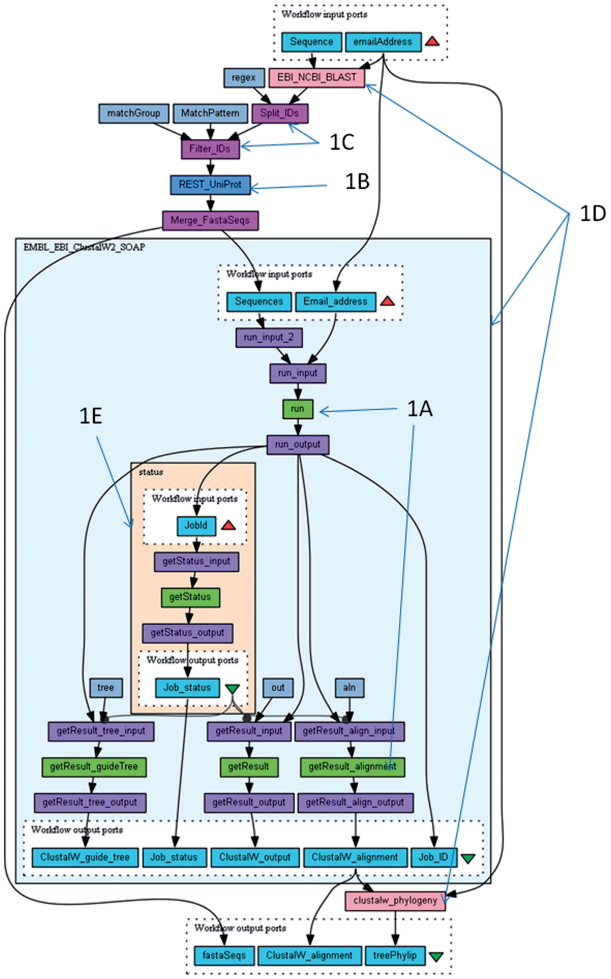


Figure 3. Sample Taverna workflow that implements Blast_Align_and_Tree (taken from [56])

6.1. Implementing Two-level Workflows for Astronomy with Taverna and COMPSs

The astrophysics community is facing a huge challenge both in terms of computing and data with the Square Kilometer Array (SKA [50]), where they expect to reach data rates in the exascale domain. They expect up to 10 exabytes of data per day, and are planning to build an exascale computing platform that can deal with this amount of data and that it is able to process it, sometimes in near real-time.

Taverna has traditionally been used in this area, however, since recently the exploitation of distributed computing infrastructures with Taverna was quite limited, and in general the exploitation of the parallelism is not the strong point of the environment. An alternative implementation of workflows, was considered in [45], with the combination of workflows at two levels: first level driven by Taverna and a second level driven by COMPSs.

COMPSs [32, 53] is a framework which aims to ease the development and execution of parallel applications for distributed infrastructures, such as Clusters and Clouds. A COMPSs application is composed of tasks, which are annotated methods. At execution time, the runtime builds a task graph that takes into account the data dependencies between tasks, and from this graph schedules and executes the tasks in the distributed infrastructure, taking also care of the required data transfers between nodes. COMPSs is written in Java, and supports applications in Java, Python and C/C++. Between the features of COMPSs, we find that the workflow can

be composed of tasks that are regular methods or web services, and that the whole COMPSs application can be published as a web service.

Another feature of COMPSs is that their applications are agnostic of the actual computing infrastructure where they are executed. This is accomplished through a component that offers different connectors, each bridging to each provider API. COMPSs can run in different Cloud providers and federation of them, and in clusters and supercomputers. COMPSs runtime also supports elasticity in clouds and federated clouds.

COMPSs applications can be exposed as web services, and internally these web services are task-based applications able to run in parallel in distributed computing platforms. These web services can then be combined with the Taverna Workbench into graphical workflows. This approach results in a two-level workflow system: at the user level, workflows are built upon web services, while those services turn out to be workflows as well at the infrastructure level. The architecture of this solution is shown in fig. 4.

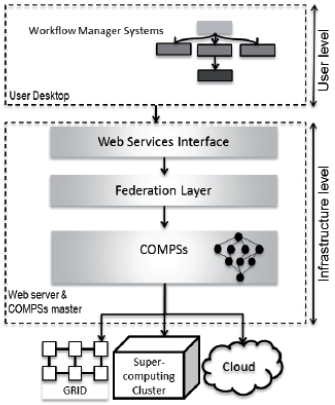


Figure 4. Two-level workflow system architecture for a astrophysics use case (taken from [45])

The cases considered compose a set of of analysis tasks of interest for some user applications of the SKA community. The focus is on the kinematical modelling of galaxies, which is applied in the study of galaxies evolution.

A common practice is to run the set of tasks with different parameters, in order to generate several models. Therefore, several workflows are executed, and later there is a manual phase from the astronomer to choose the optimal generated model. Given the large amount of data that it is foreseen to be generated by SKA, the workflows have been designed to execute the processing tasks where the data is stored.

The web services were deployed in a supercomputing cluster and in a distributed computing infrastructure (IBERGRID). The COMPSs services were configured to receive either individual sets of parameters to run a single combination of data or a list of sets of parameters in order to run multiple time the same workflow. This is easy to be implemented in COMPSs, since offers a programmatic interface, and it is also executed very fast in its runtime, while in other systems like the same Taverna Workbech, either was difficult to specify since it is not that simple to specify a loop in the graphical interface or it was not as efficient as expected.

Taverna Workbench Astronomy 2.5 was used to edit the graphical workflows, a special edition of the Taverna Workbench that includes support for building and executing astronomy workflows based on VO services through the Astrotaverna plug-in [44]. The workflows have been published in the myExperiment repository and can be accessed by the community.

7. Intelligent Workflows

Previous sections have described current approaches to manage scientific workflows and successful use cases deployed in distributed computing. With the Exascale era around the corner, the community faces a unique opportunity of implementing a new generation of intelligent workflows, which involve large simulations together with data analytics.

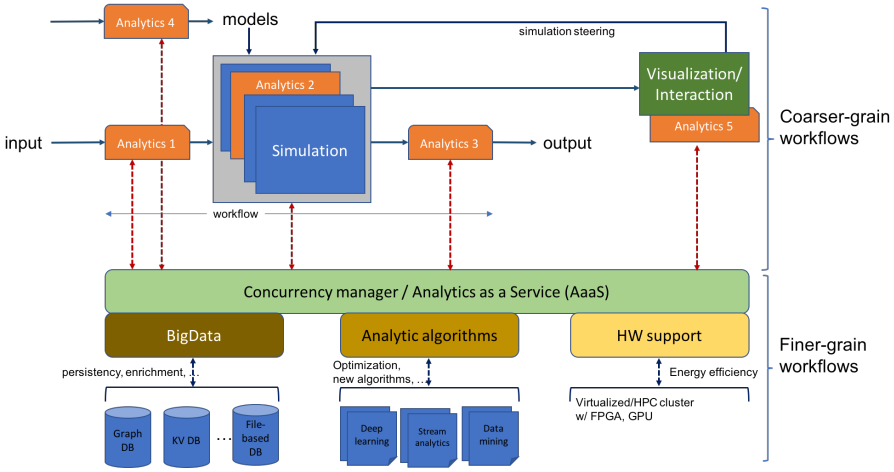


Figure 5. Architecture of new intelligent workflows

Such workflows (see fig. 5) will be composed of HPC simulations (a single task or node in the workflow may be a large MPI+X simulation involving several computing nodes), data analytics (which can be both at the input, interleaved with computation, or at the output) and visualization. The actual workflow should not be static, but dynamically instantiated according to the needs of the overall application objective. This will prevent *brute force* execution of large simulations, otherwise enabling the dynamic deployment of new simulations or computations in order to, for example, enact a finer simulation cycle since previous analytics cycle detect a given anomaly in the previous results.

At a higher level, the system should provide an end-to-end coordination layer that enables the management of dynamic workflows composed of simulations, analytics and visualization, including inputs/outputs from streams. Since graphical interfaces usually lack of enough tools to express dynamicity, a programmatic interface would be probably more appropriate. Programmatic interfaces does not only support the description of iterative constructions like conditional loops, but offer the whole expressiveness of the programming language to express complex algorithms, like optimization searches, etc. For example, PyCOMPSs [53] or Swift [55] offer programmatic/scripting interfaces.

Additionally, taking into account that some application areas may require the possibility of accepting streamed input data (from sensors or other sources of dynamic data) and streamed output data (visualization, monitoring, etc) the system should support this type of data acquisition.

This first coarser grain level of workflows will include a set of analytics, implemented as fine grain workflows. These analytics can be provided as a layer of Analytics as a Service that can be used by the workflows depending on their requirements. The analytics may implement algorithms that can be parallelized as well, but usually this type of algorithms does not show a parallelism easy to deal with traditional parallel programming models such as MPI, that is why task-based programming models seem to be a better approach to implement such services.

The services will be executed in a set of nodes of the same computing infrastructure, showing an inherent parallelism described in the form of a finer grain workflow or task-graph.

Alternatives to implement these services can be traditional Big Data programming models, such as Spark [58] or Hadoop [24], although these systems sometimes lack of the expected performance in HPC systems [16], and environments that include runtimes with more performance may be required for this purpose. Of special interest to implement these analytics can be the use of GPUs or accelerators, that have proven to be key in the implementation of fast Neural Networks used in Deep Learning [47].

The amount of data received, processed and generated with these workflows will require new solutions for storage that go beyond the traditional file systems. New storage devices such as Non-volatile RAM and storage class memories support data persistency and byte addressable access, with a performance between memories and SSDs. These devices enable the availability of data while being generated, without the need of writing the data to disk. A new type of consumer-producer applications can be designed, where the data can be stored in these persistent storage and be accessed during the execution of the producer application or after. While this data can be stored in files or databases, both are designed to use block devices, while this type of storage supports other alternatives, as direct object storage [36].

In the environment described before, persistent storage can be used to store the results of simulations. The data can be consumed by the analytic services as soon as it has been produced and the results of the analytic steps can also be stored in persistent storage, in order to be used in visualization steps or in future queries. New challenges that appear are decisions on which data should be stored in each level of the storage hierarchy, since probably the persistent layer would have less capacity, or how to perform garbage collection in such memories (since data is persistent after the execution of the applications), and its integration with the programming models, since a clean interface should be provided to the programmers.

7.1. Summary and Systems Comparison

As a summary of the paper, this subsection discusses the main features of the Workflow Management Systems (WFS) described in this paper in comparison with the new WFS architecture proposed in this section. This comparison is shown in Tab. 1.

Table 1. Workflow Management Systems features comparison

Feature / WMS	Galaxy	Kepler	Autosubmit	Taverna	Pegasus	(COMPSs	Int. Workflows
Interface	Graphical	Graphical	Script	Graphical	Textual	Programmatic	Programmatic
Parallel tasks	Limited	Yes	Yes	Limited	Yes	Yes	Yes
Dynamic workflow	No	No	No	No	Somehow	Yes	Yes
Hierarchy	No	Yes	No	Yes	Somehow	Somehow	Yes
Support for streams	No	Yes	No	Yes	No	No	Yes
Support for visualization	Yes	Yes	No	No	No	No	Yes
Support for new stor. tech.	No	No	No	No	No	Yes	Yes
Support for accelerators	No	No	No	No	Somehow	Yes	Yes

As a general comment, we believe that WMS should be generic enough to cover the requirements of different scientific communities. It is reasonable that exists solutions home-made or ad-hoc which are later adopted by more users and extended, but we consider that this should not be the best practice.

A graphical interface is sometimes preferred by non expert programmers. However, drawing large workflows that include conditional and loops can be a difficult task. Programmatical interfaces offer the flexibility and expressiveness of the programming model: the behaviour of a

complex workflow can be described with a few lines of code. These interfaces can be supported with graphical tools that visually show the obtained workflow. Also, this interface is able to naturally support dynamic workflows.

The support for parallel tasks is, in most of the current systems, limited to multi-threaded intranode tasks. Also, in some systems, although MPI tasks are supported, this support is through its interaction with the batch system sending the whole task to the queuing system. However, this is a key feature when considering workflows of HPC applications, with tasks that are MPI applications executed across multiple nodes of a cluster.

The support for hierarchy is in a similar situation in the existing systems, sometimes supported through invocation of new instances of the engine, like in Pegasus or in the current version of PyCOMPSs. This feature is very relevant in order to compose sub-workflows semantically different into larger workflows. For example, a sub-workflow may compose a set of HPC simulations, while another sub-workflow implement analytics of the results of these simulations.

Support for streaming and visualization are related features with limited support in current systems (only Kepler supports both of them), but that are key for the support of end-to-end workflows which involve inputs and outputs from multiple sources.

The support for new storage technologies and new architectures like accelerators have not been considered so far by most of the current systems, but as technology evolve the WMS should also consider them to improve performance and functionality.

Conclusions

While the scientific community has a unified view of what is a workflow, the different instances of Workflow Management Systems available for researchers have large variety: options for the interface, views on what can be a workflow tasks, types of data being exchanged by the tasks, engine complexity, computing platform, etc.

This different nature is sometimes explained by the best practices of specific communities and by the type of workflows each community requires. For example, for some communities, having an intuitive graphical interface with the possibility of editing their workflows with a simple drag and drop is essential, while for others, simplifying the access to large supercomputers where they can run large parallel applications is a must.

However, within a given scientific community WMS with similar characteristics or interoperable between them are used. This is the case of Galaxy and Taverna, for example, largely used in bioinformatics research, for which even exists a system, Tavaxy [2], that supports both systems' workflows.

While offering a single workflow management system for all scientific communities does not seem possible, we believe that interoperability between similar systems should be promoted, through common workflow description languages or interoperable interfaces. What is more, a new family of workflow management systems that enable better integration between the computation and analytics of the workflows should be designed. These systems should enable a smarter definition of the workflows, which will be more efficient in the usage of computing and storage resources, and more effective on performing the required computations and analysis that are required by the scientists. With regard the computing infrastructures, new architectures that include new computing devices (GPUs, FPGAs and other accelerators), and new storage hierarchies and technologies should be considered.

Acknowledgments

This work has been supported by the Spanish Government (SEV2015-0493), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272). This work is also supported by the Intel-BSC Exascale Lab. The Human Brain Project receives funding from the EU's Seventh Framework Programme (FP7/2007-2013) under grant agreement no 604102.

References

1. Abbott, B., Abbott, R., Adhikari, R., Ajith, P., Allen, B., Allen, G., Amin, R., Anderson, S., Anderson, W., Arain, M., et al.: Ligo: the laser interferometer gravitational-wave observatory. Reports on Progress in Physics 72(7), 076901 (2009), DOI: 10.1088/0034-4885/72/7/076901

2. Abouelhoda, M., Issa, S.A., Ghanem, M.: Tavaxy: Integrating taverna and galaxy workflows with cloud computing support. BMC bioinformatics 13(1), 77 (2012)

3. Afgan, E., Baker, D., van den Beek, M., Blankenberg, D., Bouvier, D., Čech, M., Chilton, J., Clements, D., Coraor, N., Eberhard, C., et al.: The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. Nucleic acids research p. gkw343 (2016)

4. Afgan, E., Chapman, B., Taylor, J.: Cloudman as a platform for tool, data, and analysis distribution. BMC Bioinformatics 13(1), 315 (2012), DOI: 10.1186/1471-2105-13-315

5. Afgan, E., Coraor, N., Chilton, J., Baker, D., Taylor, J., Team, T.G.: Enabling cloud bursting for life sciences within galaxy. Concurrency and Computation: Practice and Experience 27(16), 4330–4343 (2015), cPE-15-0018.R1

6. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on. pp. 423–424. IEEE (2004)

7. Amstutz, P., Crusoe, M.R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leehr, D., Mnager, H., Nedeljkovich, M., Scales, M., Soiland-Reyes, S., Stojanovic, L.: Common Workflow Language, v1.0. Tech. rep. (3 2016), `https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156`, DOI: 10.6084/m9.figshare.3115156.v2

8. Barseghian, D., Altintas, I., Jones, M.B., Crawl, D., Potter, N., Gallagher, J., Cornillon, P., Schildhauer, M., Borer, E.T., Seabloom, E.W., Hosseini, P.R.: Workflows and extensions to the kepler scientific workflow system to support environmental sensor data access and analysis. Ecological Informatics 5(1), 42 – 50 (2010), `http://www.sciencedirect.com/science/article/pii/S1574954109000673`, special Issue: Advances in environmental information management

9. Berthold, M.R., Cebron, N., Dill, F., Gabriel, T.R., Kötter, T., Meinl, T., Ohl, P., Sieb, C., Thiel, K., Wiswedel, B.: KNIME: The Konstanz Information Miner. In: Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007). Springer (2007)

10. BioExcel website. Web page at http://www.bioexcel.eu, accessed: 2017-02-15

11. Cabellos, L., Campos, I., del Castillo, E.F., Owsiak, M., Palak, B., Pciennik, M.: Scientific workflow orchestration interoperating htc and hpc resources. Computer Physics Communications 182(4), 890 – 897 (2011), http://www.sciencedirect.com/science/article/pii/S0010465510005096

12. Coppens, F., Corpas, M.: Recommendation for actions on Galaxy for ELIXIR HoNs. available at https://www.elixir-europe.org/about/groups/galaxy-wg, accessed: 2017-02-15

13. Coster, D.P., Basiuk, V., Pereverzev, G., Kalupin, D., Zagorksi, R., Stankiewicz, R., Huynh, P., Imbeaux, F., et al.: The European Transport Solver. IEEE Transactions on Plasma Science 38(9), 2085–2092 (2010)

14. Deelman, E., Gannon, D., Shields, M., Taylor, I.: Workflows and e-science: An overview of workflow system features and capabilities. Future Generation Computer Systems 25(5), 528 – 540 (2009), http://www.sciencedirect.com/science/article/pii/S0167739X08000861

15. Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., da Silva, R.F., Livny, M., et al.: Pegasus, a workflow management system for science automation. Future Generation Computer Systems 46, 17–35 (2015)

16. Ekanayake, S., Kamburugamuve, S., Wickramasinghe, P., Fox, G.C.: Java thread and process performance for parallel machine learning on multicore hpc clusters. In: Proceedings of the 2016 IEEE International Conference on Big Data (2016)

17. Elixir website. Web page at https://www.elixir-europe.org, accessed: 2017-02-15

18. Building an European Reseach Community through Interoperable Workflows and Data. Web page at http://www.erflow.eu, accessed: 2017-02-15

19. Erwin, D.W., Snelling, D.F.: Unicore: A grid computing environment. In: European Conference on Parallel Processing. pp. 825–834. Springer (2001)

20. European Consortium for the Development of Fusion Energy. Web page at https://www.euro-fusion.org, accessed: 2017-02-15

21. Fahringer, T., Prodan, R., Duan, R., Nerieri, F., Podlipnig, S., Qin, J., Siddiqui, M., Truong, H.L., Villazon, A., Wieczorek, M.: Askalon: A grid application development and computing environment. In: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing. pp. 122–131. IEEE Computer Society (2005)

22. Falchetto, G.L., Coster, D., Coelho, R., Scott, B., Figini, L., Kalupin, D., Nardon, E., Nowak, S., Alves, L.L., Artaud, J.F., et al.: The european integrated tokamak modelling (itm) effort: achievements and first physics results. Nuclear Fusion 54(4), 043018 (2014)

23. Goble, C.A., Bhagat, J., Aleksejevs, S., Cruickshank, D., Michaelides, D., Newman, D., Borkum, M., Bechhofer, S., Roos, M., Li, P., et al.: myexperiment: a repository and social network for the sharing of bioinformatics workflows. Nucleic acids research 38(suppl 2), W677–W682 (2010)

24. Apache Hadoop. Web page at http://hadoop.apache.org/ ((Date of last access: 15th November, 2016))

25. Hospital, A., Montras, A., Soiland-Reyes, S., Bonvin, A., Melquiond, A., Gelpí, J.L., Lezzi, D., Newhouse, S., Dianes, J.A., Abraham, M., Apostolov, R., Ippoliti, E., Carter, A., White, D.J.: D2.1 State of the art and gap analysis. Tech. rep., BioExcel deliverable (2016)

26. Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M.R., Li, P., Oinn, T.: Taverna: a tool for building and running workflows of services. Nucleic acids research 34(suppl 2), W729–W732 (2006)

27. Imbeaux, F., Pinches, S., Lister, J., Buravand, Y., Casper, T., Duval, B., Guillerminet, B., Hosokawa, M., Houlberg, W., Huynh, P., Kim, S., Manduchi, G., Owsiak, M., Palak, B., Plociennik, M., Rouault, G., Sauter, O., Strand, P.: Design and first applications of the iter integrated modelling & analysis suite. Nuclear Fusion 55(12), 123006 (2015), http://stacks.iop.org/0029-5515/55/i=12/a=123006

28. InfraStructure for the European Network for the Earth System Modelling. Web page at https://is.enes.org, accessed: 2017-02-15

29. The Kepler Project. Web page at https://kepler-project.org, accessed: 2017-02-15

30. Kranzlmüller, D., de Lucas, J.M., Öster, P.: The european grid initiative (egi). In: Remote Instrumentation and Virtual Laboratories, pp. 61–66. Springer (2010)

31. Laure, E., Edlund, A., Pacini, F., Buncic, P., Barroso, M., Di Meglio, A., Prelz, F., Frohner, A., Mulmo, O., Krenek, A., et al.: Programming the grid with glite. Tech. rep. (2006)

32. Lordan, F., Tejedor, E., Ejarque, J., Rafanell, R., Alvarez, J., Marozzo, F., Lezzi, D., Sirvent, R., Talia, D., Badia, R.M.: ServiceSs: An Interoperable Programming Framework for the Cloud. Journal of Grid Computing 12(1), 67–91 (2014)

33. Manubens-Gil, D., Vegas-Regidor, J., Matthews, D., Shin, M.: Assesment report on autosubmit, cylc and ecflow. Tech. rep. (2016), https://earth.bsc.es/wiki/lib/exe/fetch.php?media=tools:isenes2_d93_v1.0_mp.pdf

34. Manubens-Gil, D., Vegas-Regidor, J., Prodhomme, C., Mula-Valls, O., Doblas-Reyes, F.J.: Seamless management of ensemble climate prediction experiments on hpc platforms. In: High Performance Computing & Simulation (HPCS), 2016 International Conference on. pp. 895–900. IEEE (2016)

35. Manubens-Gila, D., Vegas-Regidora, J., Acostaa, M.C., Prodhommea, C., Mula-Vallsa, O., Serradell-Marondaa, K., Doblas-Reyes, F.J.: Autosubmit: a versatile tool for managing Earth system models on HPC platforms. Future Generation Computer Systems submited (2016)

36. Marti, J., Gasull, D., Queralt, A., Cortes, T.: Towards DaaS 2.0: Enriching data models. In: Proceedings - 2013 IEEE 9th World Congress on Services, SERVICES 2013. pp. 349–355. IEEE, IEEE (jun 2013), DOI: 10.1109/SERVICES.2013.59

37. McLennan, M., Clark, S., Deelman, E., Rynge, M., Vahi, K., McKenna, F., Kearney, D., Song, C.: Hubzero and pegasus: integrating scientific workflows into science gateways. Concurrency and Computation: Practice and Experience (2014), DOI: 10.1002/cpe.3257

38. National Virtual Observatory. Web page at `http://us-vo.org`, accessed: 2017-02-15

39. Oliver, H.J.: Cylc (the cylc suite engine). Tech. rep. (2016)

40. Pordes, R., Petravick, D., Kramer, B., Olson, D., Livny, M., Roy, A., Avery, P., Blackburn, K., Wenaus, T., Würthwein, F., et al.: The open science grid 78(1), 012057 (2007)

41. Price, B.: Frank and lillian gilbreth and the manufacture and marketing of motion study, 1908-1924. Business and economic history pp. 88–98 (1989)

42. Pronk, S., Larsson, P., Pouya, I., Bowman, G.R., Haque, I.S., Beauchamp, K., Hess, B., Pande, V.S., Kasson, P.M., Lindahl, E.: Copernicus: A new paradigm for parallel adaptive molecular dynamics. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 60:1–60:10. SC '11, ACM, New York, NY, USA (2011), DOI: 10.1145/2063384.2063465

43. Ptolemaeus, C. (ed.): System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014), `http://ptolemy.org/books/Systems`

44. Ruiz, J., Garrido, J., Santander-Vela, J., Sánchez-Expósito, S., Verdes-Montenegro, L.: Astrotavernabuilding workflows with virtual observatory services. Astronomy and Computing 7, 3–11 (2014)

45. Sánchez-Expósito, S., Martín, P., Ruíz, J.E., Verdes-Montenegro, L., Garrido, J., Sirvent, R., Falcó, A.R., Badia, R., Lezzi, D.: Web services as building blocks for science gateways in astrophysics. Journal of Grid Computing 14(4), 673–685 (2016)

46. Southern California Earthquake Center. Web page at `http://scec.org/`, accessed: 2017-02-15

47. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Networks 61, 85 – 117 (2015), `http://sciencedirect.com/science/article/pii/S0893608014002135`

48. Scott, B.D., Weinberg, V., Hoenen, O., Karmakar, A., Fazendeiro, L.: Scalability of the plasma physics code gem. arXiv preprint arXiv:1312.1187 (2013)

49. SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs. Web page at `http://www.shiwa-workflow.eu/`, accessed: 2017-02-15

50. Square Kilometre Array. Web page at `https://www.skatelescope.org`, accessed: 2017-02-15

51. Sloggett, C., Goonasekera, N., Afgan, E.: Bioblend: automating pipeline analyses within galaxy and cloudman. Bioinformatics 29(13), 1685–1686 (2013)

52. The Principles of Scientific Management. The Mathematics Teacher 4(1), 44–44 (1911), http://www.jstor.org/stable/27949698

53. Tejedor, E., Becerra, Y., Alomar, G., Queralt, A., Badia, R.M., Torres, J., Cortes, T., Labarta, J.: Pycompss: Parallel computational workflows in python. International Journal of High Performance Computing Applications (2015)

54. Galaxy Tool Sheed. Web page at https://toolshed.g2.bx.psu.edu, accessed: 2017-02-15

55. Wilde, M., Hategan, M., Wozniak, J.M., Clifford, B., Katz, D.S., Foster, I.: Swift: A language for distributed parallel scripting. Parallel Computing 37(9), 633–652 (2011)

56. Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., et al.: The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. Nucleic acids research p. gkt328 (2013)

57. Yu, J., Buyya, R.: A taxonomy of workflow management systems for grid computing. Journal of Grid Computing 3(3-4), 171–200 (2005)

58. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster Computing with Working Sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. HotCloud'10, USENIX Association, Berkeley, CA, USA (2010)

DOI: 10.14529/jsfi170103

A Survey: Runtime Software Systems for High Performance Computing

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

High Performance Computing 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. These methods they are principally coarse 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 future 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, High Performance Computing.

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

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

has been made in the arena of HPC. This is not absolutely the case as some modest amount 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 neither 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 imposes 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 2 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 3 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 4 delivers brief descriptions of a larger number of runtimes, including some limited to single node SMPs like OpenMP. To summarize as well as compare and contrast these runtimes, tables are provided in Section 5. 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.

1. 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 [5] 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%. However, routinely 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 are 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 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 [4] 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.

2. Key Functionality

Runtime systems for HPC vary widely in the specific details of their semantics and software implementation. But they have 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.

2.1. Multi-Threading

Almost all runtime systems in use have some form of threaded execution, whether directly employing POSIX Pthreads [23] 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 [18]) 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 [24]) 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).

2.2. 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 ex-

plicitly 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.

2.3. 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" [19] 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 [8] 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.

2.4. 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 [1] 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.

3. Major Runtime Exemplars

3.1. Charm++

Charm++ [10, 22] 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 [29] 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 synchro-

nized 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.

3.2. OCR

The Open Community Runtime (OCR) [17] 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 works 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 [27] (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 [6], R-Stream compiler [12], Habanero-C [25], and Habanero-UPC++ [11] libraries.

3.3. 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 [36], 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 [37] 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.

3.4. Legion

Legion [15] 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 work 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 [35] and CUDA and is still in active development. Legion has demonstrated scalability in combustion simulations using

the S3D mini-application [7] 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 generated 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 that have already been written. Only the mapping has to change when moving code between different machine architectures in order to optimize performance. Mapping changes such as altering 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.

3.5. OmpSs

OmpSs [41] is a task based programming model developed at the Barcelona Supercomputing Center that aims to 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 [16] programming model names. A runtime implementation of OmpSs is provided through the Nanos++ [39] runtime system research tool which provides user-level threads, synchronization support, and heterogeneity support and through the Mercurium compiler [38] 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 tasks are assigned 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.

3.6. PaRSEC

The Parallel Runtime Scheduling and Execution Controller (PaRSEC) framework [32] 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 [2]. 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 an effectively 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 [26], 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 [3]. PaRSEC also features a resilience policy with data logging and checkpointing integrated into the runtime model.

4. Other Runtimes

4.1. Qthreads

Sandia's Qthreads library [20] 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 resources, 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 [14]. 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.

4.2. DARMA

The Distributed Asynchronous Resilient Models and Applications (DARMA) co-design programming model [21] 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.

4.3. OpenMP and OpenACC

OpenMP [31], along with the OpenACC [30] 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.

4.4. Cilk

The Cilk [13] family with its variants Cilk++ and Cilk Plus [9] 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.

4.5. TBB

Intel TBB [42] 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.

4.6. Argobots

Argobots [33] 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 [34] which utilizes Argobots to provide OpenMP performance specialized for fine-grained application execution.

4.7. XcalableMP

XcalableMP (XMP) [28] 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 [40] at RIKEN and the University of Tsukuba.

5. Summary Table

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

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 require similar forms of adaptive control. Applications like AMR 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

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.

is possible that the near term explorations of runtime software for optimizing applications will ultimately drive changes in the HPC hardware system design.

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

References

1. Baker, H.C., Hewitt, C.: The incremental garbage collection of processes. In: SIGART Bull. pp. 55–59. ACM, New York, NY, USA (August 1977), DOI: 10.1145/872736.806932

2. Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., Dongarra, J.: PTG: An abstraction for unhindered parallelism. In: 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing. pp. 21–30 (Nov 2014), DOI: 10.1109/wolfhpc.2014.8

3. Danalis, A., Jagode, H., Bosilca, G., Dongarra, J.: Parsec in practice: Optimizing a legacy chemistry application through distributed task-based execution. In: 2015 IEEE International Conference on Cluster Computing. pp. 304–313 (Sept 2015), DOI: 10.1109/cluster.2015.50

4. Dennard, R.H., Gaensslen, F., Yu, H.N., Rideout, L., Bassous, E., LeBlanc, A.: Design of ion-implanted MOSFET's with very small physical dimensions. IEEE Journal of Solid State Circuits 9(5) (October 1974), DOI: 10.1109/jssc.1974.1050511

5. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK benchmark: past, present and future. Concurrency and Computation, Practice and Experience 15(9) (July 2003), DOI: 10.1002/cpe.728

6. Fraguela, B.B., Bikshandi, G., Guo, J., Garzarán, M.J., Padua, D., Von Praun, C.: Optimization techniques for efficient HTA programs. Parallel Comput. 38(9), 465–484 (Sep 2012), DOI: 10.1016/j.parco.2012.05.002

7. Grout, R., Sankaran, R., Levesque, J., Woolley, C., Posy, S., Chen, J.: 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

8. Hewitt, C., Baker, H.G.: Actors and continuous functionals. Tech. rep., Cambridge, MA, USA (1978)

9. 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

10. Kale, L.V., Krishnan., S.: Charm++: Parallel programming with message-driven objects. In: Wilson, G.V., Lu, P. (eds.) Parallel Programming using C++, pp. 175–213. MIT Press (1996)

11. Kumar, V., Zheng, Y., Cavé, V., Budimlić, Z., Sarkar, V.: HabaneroUPC++: A compiler-free PGAS library. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. pp. 5:1–5:10. PGAS '14, ACM, New York, NY, USA (2014), DOI: 10.1145/2676870.2676879

12. Lethin, R., Leung, A., Meister, B., Schweitz, E.: 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

13. MIT: Cilk 5.4.6 Reference Manual (1998), http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf

14. Schneider, T., Hoefler, T., Grant, R.E., Barrett, B.W., Brightwell, R.: Protocols for fully offloaded collective operations on accelerated network adapters. In: 42nd International Conference on Parallel Processing. pp. 593–602 (Oct 2013)

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

16. Subotić, V., Brinkmann, S., Marjanovi, V., Badia, R.M., Gracia, J., Niethammer, C., Ayguade, E., Labarta, J., Valero, M.: Programmability and portability for exascale: Top down programming methodology and tools with StarSs. Journal of Computational Science 4(6), 450 – 456 (2013), http://www.sciencedirect.com/science/article/pii/S1877750313000203, scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011

17. Tim, M., Romain, C.: 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

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

19. Von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active messages: A mechanism for integrated communication and computation. Proceedings of The 19th Annual International Symposium on Computer Architecture, 1992 pp. 256–266 (1992), DOI: 10.1109/isca.1992.753322

20. Wheeler, K., Murphy, R., Thain, D.: 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), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4536359

21. Wilke, J., Hollman, D., Slattengren, N., Lifflander, J., Kolla, H., Rizzi, F., Teranishi, K., Bennett, J.: DARMA 0.3.0-alpha specification (March 2016), version 0.3.0-alpha, SANDIA Report SAND2016-5397

22. The Charm++ parallel programming system manual, version 6.7.1, http://charm.cs.illinois.edu/manuals/pdf/charm++.pdf, accessed: 2017-02-15

23. IEEE Standard for Information Technology – Portable Operating System Interface (POSIX®). IEEE Standard (2008), http://standards.ieee.org/findstds/standard/1003.1-2008.html, accessed: 2017-02-15

24. SWARM (SWift Adaptive Runtime Machine) (2011), white paper, http://www.etinternational.com/files/2713/2128/2002/ETI-SWARM-whitepaper-11092011.pdf, accessed: 2017-02-15

25. Habanero-C (2013), website, https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C, accessed: 2017-02-15

26. DPLASMA: distributed parallel linear algebra software for multicore architectures (April 2014), version 1.2.0 http://icl.utk.edu/dplasma/, accessed: 2017-02-15

27. Intel® concurrent collections C++ API (June 2014), website, https://icnc.github.io/api/index.html, accessed: 2017-02-15

28. XcalableMP: a directive-based language extension for scalable and performance-aware parallel programming (Nov 2014), version 1.2.1 http://www.xcalablemp.org/, accessed: 2017-02-15

29. MPI: A Message-Passing Interface Standard (June 2015), specification document, http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, accessed: 2017-02-15

30. The OpenACC application programming interface (October 2015), version 2.5, http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf, accessed: 2017-02-15

31. OpenMP application programming interface (November 2015), version 4.5, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, accessed: 2017-02-15

32. 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, accessed: 2017-02-15

33. Argobots: a lightweight low-level threading/tasking framework (Nov 2016), version 1.0a1 http://www.argobots.org/, accessed: 2017-02-15

34. BOLT: a lightning-fast OpenMP implementation (Nov 2016), version 1.0a1 http://www.mcs.anl.gov/bolt/, accessed: 2017-02-15

35. GASNet low-level networking layer (Oct 2016), version 1.28.0, https://gasnet.lbl.gov/, accessed: 2017-02-15

36. HPX (July 2016), version 0.9.99, http://stellar.cct.lsu.edu/, accessed: 2017-02-15

37. HPX-5 (Nov 2016), version 4.0.0 http://hpx.crest.iu.edu/, accessed: 2017-02-15

38. The Mercurium source-to-source compilation infrastructure (June 2016), version 2.0.0 https://pm.bsc.es/mcxx, accessed: 2017-02-15

39. The Nanos++ runtime system (June 2016), version 0.10 https://pm.bsc.es/nanox, accessed: 2017-02-15

40. Omni (Nov 2016), version 1.1.0 http://omni-compiler.org/, accessed: 2017-02-15

41. The OmpSs programming model (June 2016), version 16.06 https://pm.bsc.es/ompss, accessed: 2017-02-15

42. Intel® Threading Building Blocks (Intel® TBB) (2017), website, http://www.threadingbuildingblocks.org, accessed: 2017-02-15

DOI: 10.14529/jsfi170104

xSDK Foundations: Toward an Extreme-scale Scientific Software Development Kit

***Roscoe Bartlett*[1], *Irina Demeshko*[2], *Todd Gamblin*[3], *Glenn Hammond*[1], *Michael Heroux*[1], *Jeffrey Johnson*[4], *Alicia Klinvex*[1], *Xiaoye Li*[5], *Lois Curfman McInnes*[6], *J. David Moulton*[2], *Daniel Osei-Kuffuor*[3], *Jason Sarich*[6], *Barry Smith*[6], *Jim Willenbring*[1], *Ulrike Meier Yang*[3]**

Extreme-scale computational science increasingly demands multiscale and multiphysics formulations. Combining software developed by independent groups is imperative: no single team has resources for all predictive science and decision support capabilities. Scientific libraries provide high-quality, reusable software components for constructing applications with improved robustness and portability. However, without coordination, many libraries cannot be easily composed. Namespace collisions, inconsistent arguments, lack of third-party software versioning, and additional difficulties make composition costly.

The Extreme-scale Scientific Software Development Kit (xSDK) defines community policies to improve code quality and compatibility across independently developed packages (hypre, PETSc, SuperLU, Trilinos, and Alquimia) and provides a foundation for addressing broader issues in software interoperability, performance portability, and sustainability. The xSDK provides turnkey installation of member software and seamless combination of aggregate capabilities, and it marks first steps toward extreme-scale scientific software ecosystems from which future applications can be composed rapidly with assured quality and scalability.

Keywords: xSDK, Extreme-scale scientific software development kit, numerical libraries, software interoperability, sustainability.

1. Software Challenges for Extreme-scale Science

Extreme-scale architectures provide unprecedented resources for scientific discovery. At the same time, the computational science and engineering (CSE) community faces daunting productivity and sustainability challenges for parallel application development [1, 12, 13, 21]. Difficulties include increasing complexity of algorithms and computer science techniques required by coupled multiscale and multiphysics applications. Further complications come from the imperative of portable performance in the midst of dramatic and disruptive architectural changes on the path to exascale, the realities of large legacy code bases, and human factors arising in distributed multidisciplinary research teams pursuing leading edge parallel performance. Moreover, new architectures require fundamental algorithm and software refactoring, while at the same time demand is increasing for greater reproducibility of simulation and analysis results for predictive science.

This confluence of challenges brings with it a unique opportunity to fundamentally change how scientific software is designed, developed, and sustained. The demands arising from so many challenges force the CSE community to consider a broader range of potential solutions. It is this setting that makes possible a collaborative effort to establish a scientific software ecosystem of

[1]Sandia National Laboratories, USA
[2]Los Alamos National Laboratory, New-Mexico, USA
[3]Lawrence Livermore National Laboratory, Livermore, USA
[4]Salesforce, San-Francisco, USA
[5]Lawrence Berkeley National Laboratory, Berkeley, USA
[6]Argonne National Laboratory, Lemont, USA

reusable libraries and community policies to guide common adoption of practices, tools, and infrastructure. Incremental change is not a viable option, so migration to a new model for CSE software is possible.

The xSDK has emerged as a first step toward a new ecosystem, where application codes are composed via interfaces from a common base of reusable components more than they are developed from a clean slate or derived from monolithic code bases. To the extent that this compositional approach can be reliably used, new CSE applications can be created more rapidly, with greater robustness and scalability, by smaller teams of scientists, enabling them to focus more attention on obtaining science results than on the incendentals of their computing environment.

1.1. Related Work

The scientific software community has a rich tradition of defining *de facto* standards for collections of capabilities. EISPACK [11, 23], LINPACK [7], BLAS [8, 9, 16, 17], and LAPACK [2] delivered a sound foundation for numerical linear algebra in libraries and applications. Commercial entities such as the Numerical Algorithms Group (NAG) [26], the Harwell Subroutine Library (HSL) [30] and IMSL have provided high quality, unified software capabilities to users for decades.

More recently, the TOPS [14], ITAPS [5], and FASTMath [6] SciDAC institutes brought together developers of large-scale scientific software libraries. While these libraries were independently developed by distinct teams and version support lacked coordination, the collaborations sparked exchange of experiences and discussion of practices that avoided potential pitfalls and facilitated the combined use of the libraries [19] as needed by scientific teams. Prior efforts to provide interoperability between solver libraries can be found in PETSc [3], which allows users to access libraries such as hypre [25] and SuperLU [28] by using the PETSc interface, sparing users the effort to rebuild their problems through hypre's or SuperLU's interfaces. Trilinos [31], a collection of self-contained software packages, also provides ways for users to gain uniform access to third-party scientific libraries.

2. xSDK Vision

The complexity of application codes is steadily increasing due to more sophisticated scientific models and the continuous emergence of new high-performance computers, making it crucial to develop software libraries that provide needed capabilities and continue to adapt to new computer architectures. Each library is complex and requires different expertise. Without coordination, and in service of distinct user communities, this circumstance has led to difficulties when building application codes that use 8 or 10 different libraries, which in turn might require additional libraries or even different versions of the same libraries.

The xSDK represents a different approach to coordinating library development and deployment. Prior to the xSDK, scientific software packages were cohesive with a single team effort, but not across these efforts. The xSDK goes a step further by developing community policies followed by each independent library included in the xSDK. This policy-driven, coordinated approach enables independent development that still results in compatible and composable capabilities.

The initial xSDK project is the first step toward a comprehensive software ecosystem. As shown in Figure 1, the vision of the xSDK is to provide infrastructure for and interoperability of a collection of related and complementary software elements—developed by diverse, indepen-

dent teams throughout the high-performance computing (HPC) community—that provide the building blocks, tools, models, processes, and related artifacts for rapid and efficient development of high-quality applications. Our long-term goal is to make the xSDK a turnkey standard software ecosystem that is easily installed on common computing platforms, and can be assumed as available on any leadership computing system in the same way that BLAS and LAPACK are available today.

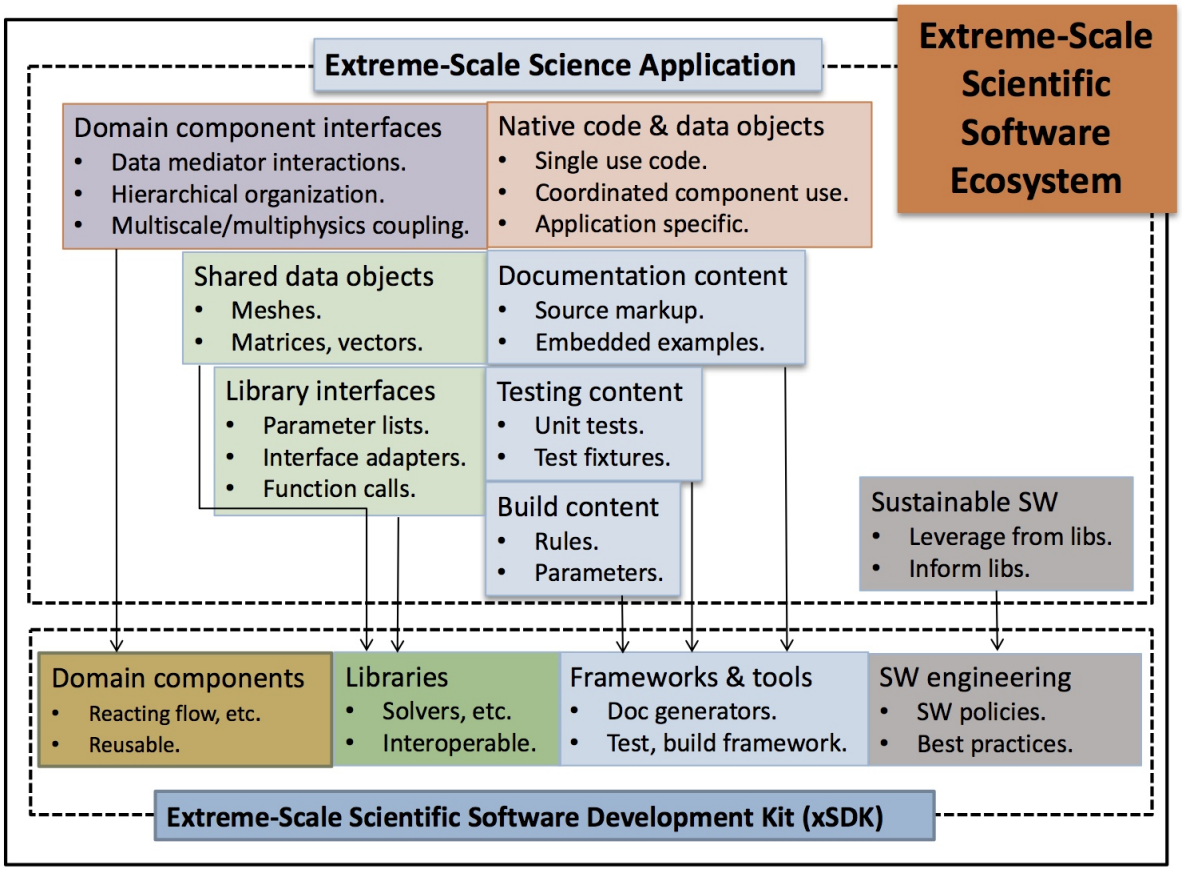


Figure 1. The xSDK intends to provide the foundation for a modern extreme-scale scientific software ecosystem, where application development is accomplished by composition of high-quality, reusable software components rather than by tangential use of libraries. Application developers produce a small portion of custom code that expresses the particular purpose of the software and then gain the bulk of functionality by parameterized use of xSDK components and libraries, which are developed by diverse, independent groups throughout the community. xSDK frameworks for documentation, testing, and code quality, as well as established software policies and best practices, can be adapted and adopted as appropriate by the application developers to provide compatible, high-quality, and sustainable software. As we move toward this new ecosystem, application development times from first concept to scalable production code should drop dramatically. Success hinges on the quality, interoperability, usability, and diversity of xSDK capabilities and our ability to deliver the xSDK to domain scientists

2.1. Elements of an Extreme-scale Scientific Software Ecosystem

Rapid, efficient production of high-quality, sustainable applications is best accomplished using a rich collection of reusable libraries, tools, lightweight frameworks, and defined software methodologies, developed by a community of scientists who are striving to identify, adapt, and adopt best practices in software engineering. Although the software engineering community has ongoing debate about the precise meaning of terms, we define the basic elements of a scientific software ecosystem to include:

- **Library:** High-quality, encapsulated, documented, tested and multi-use software that is incorporated into the application and used as native source functionality. Libraries can provide *control inversion* via abstract interfaces, call-backs, or similar techniques such that user-defined functionality can be invoked by the library, e.g., a user-defined sparse matrix multiplication routine. Libraries can also provide factories that facilitate construction of specific objects that are related by a base type and later used as an instance of the base type. Libraries can include domain-specific software components that are designed to be used by more than one application.
- **Domain component:** Reusable software that is intended for modest reuse across applications in the same domain. Although this kind of component is a library, the artifacts and processes needed to support a component are somewhat different than for a broadly reusable library.
- **Framework:** A software environment that implements specific design patterns and permits the user to insert custom content. Frameworks include documentation, build (compilation), and testing environments. These frameworks are lightweight and general purpose. Other frameworks, such as multiphysics, are considered separately, built on top of what we describe here.
- **Tool:** Software that exists outside of applications, used to improve quality, efficiency, and cost of developing and maintaining applications and libraries.
- **Software development kit (SDK):** A collection of related and complementary software elements that provide the building blocks, libraries, tools, models, processes, and related artifacts for rapid and efficient development of high-quality applications.

Given these basic elements, we define an application code as the following composition:

- **Native data and code:** Every application will have a primary routine (often a main program) and its own collection of source code and private data. Historically, applications have been primarily composed of native source and data, using libraries for a small portion of functionality, such as solvers. We foresee a decrease in the amount of native code required to develop an application by extracting and transforming useful native code into libraries and domain components, making it available to other applications.
- **Component and library function calls:** Some application functionality is provided by invoking library functions. We expect to increase usage of libraries as a part of our efforts.
- **Library interface adapters:** Advanced library integration often involves invoking the control inversion facilities of the library in order to incorporate application-specific knowledge. In the case of sensitivity analysis, embedded optimization, and related analyses, control inversion via these adapters is essential in order to permit the solver to invoke the application with specific input data.

- **Component and library parameter lists:** Libraries tend to provide a broad collection of functionality for which parameters must be set.
- **Shared component and library data:** Most libraries require the user to provide non-trivial data objects, such as meshes or sparse matrices, and may provide functions to assist the application in constructing these objects. Unlike parameter list definitions, which represent a narrow interface dependency between the application and library, application-library data interfaces can be very complicated.
- **Documentation, build, and testing content:** The application-specific text, data, and source used by the documentation, build, and testing frameworks to produce the derived software documentation, compilation, and test artifacts.

3. xSDK Approach

The xSDK approach to developing software has two distinguishing features from previous efforts in the scientific computing community:

- **Peer-to-peer interoperability:** Some previous efforts[7] attempted to use additional abstraction layers that would hide differences in the underlying packages. The xSDK approach uses the existing extensibility features of the libraries to enable peer-to-peer access of capabilities at various levels of interoperability through the native interfaces of the packages. For example, if a user has already integrated PETSc data structures into their code, the xSDK approach preserves that approach, but permits use of capabilities in hypre, SuperLU, and Trilinos with PETSc.
- **Software policies:** Most existing scientific software efforts rely on close collaboration of a single team in order to assure that collective efforts are compatible and complementary. The xSDK relies instead on *policies* that promote compatibility and complementarity of independently developed software packages. By specifying only certain expectations for how software is designed, implemented, documented, supported, and installed, the xSDK enables independent development of separate packages, while still ensuring complementarity and composability.

The xSDK can assure interoperability and compliance with community policies because the leaders and developers of xSDK packages are members of the xSDK community. If interface changes are required in a package or a version of a third-party solver needs to be updated, these changes will be made in the member package. For example, in order for Trilinos and PETSc to use the same version of SuperLU and hypre, the Trilinos and PETSc developers commit to agreeing on changes to Trilinos and PETSc that are needed for compatibility. Similarly, changes to interfaces for interoperability and inversion of control (see the next Section 3.1) are done within the xSDK packages, and regularly tested for regressions. xSDK interoperability is possible because of the commitment of xSDK member package development teams.

[7] A notable example is the Equation Solver Interface (ESI), which defined an abstraction layer to present a common client interface to distinct software products. The challenge of this approach is that the unique features of the underlying products were difficult to access. The very use of a common abstraction reduced the usability of these products.

3.1. xSDK Library Interoperability

A fundamental objective of the xSDK project is to provide interoperability layers among hypre, PETSc, SuperLU, and Trilinos packages, as appropriate, with the ultimate goal of making all mathematically meaningful interoperabilities possible in order to fully support exascale applications.

Software library interoperability refers to the ability of two or more libraries to be used together in an application, without special effort by the user [18]. For simplicity, we discuss interoperability between two libraries; extension to three or more libraries is conceptually straightforward. Depending on application needs, various levels of interoperability can be considered:

- *Interoperability level 1*: both libraries can be used (side by side) in an application
- *Interoperability level 2*: both libraries can exchange data (or control data) with each other
- *Interoperability level 3*: each library can call the other library to perform unique computations

The simplest case (interoperability level 1) occurs when an application needs to call two distinct libraries for different functionalities (for example, an MPI library for message-passing communication and HDF5 for data output). As discussed in [19, 20], even this basic interoperability requires consistency among libraries to be used in the same application, in terms of compiler, compiler version/options, and third-party capabilities. If both libraries have a dependency on a common third party, the libraries must be able to use a single common instance of it. For example, more than one version of the popular SuperLU linear solver library exists, and interfaces have evolved. If two libraries both use SuperLU, they must be able to work with the same version of SuperLU. In practice, installing multiple independently developed packages together can be a tedious trial-and-error process. The definition and implementation of xSDK community policies standards have overcome this difficulty for xSDK-compatible packages.

Interoperability level 2 builds on level 1 by enabling conversion, or encapsulation, and exchange of data between libraries. This level can simplify use of libraries in sequence by an application. In this case, the libraries themselves are typically used without internal modification to support the interoperability. Future work on node-level resource management is essential to support this deeper level of software interoperability for emerging architectures.

Interoperability level 3 builds on level 2 by supporting the use of one library to provide functionality on behalf of another library. This *integrated execution* provides significant value to application developers because they can access capabilities of additional libraries through the familiar interfaces of the first library.

The remainder of this section discusses proposed work on integrated execution, where our guiding principles are to provide interoperability that is intuitive and easy to use, and to expose functionality of each library where feasible.

Control inversion. Interoperability between two (or more) existing library components can be achieved by one of two basic mechanisms: (i) create an abstraction layer that sits on top of both components to act as an intermediary between the user and both components or (ii) permit users to write directly to the interface of one component and provide peer-level interoperability between the two components. For example, consider the matrix construction capabilities in PETSc and Trilinos. Both libraries provide extensive support for piecewise construction of sparse matrices, as needed for building objects in applications based on finite elements/volumes/differences. It would be possible, in principle, to create a top-level abstraction

layer that could be used to build a sparse matrix or other data objects for PETSc or Trilinos, depending on an input option to select either target. Alternatively, the user can construct the data object by using the PETSc or Trilinos functions directly, and then we can create adapters in Trilinos and PETSc to wrap the respective matrix object and make it behave like one of its own.

Although the first approach may seem attractive, it is difficult to develop in a sustainable and effective way. PETSc and Trilinos data object construction processes are targeted to specific programming, language, and usage models. The differences in approach may appear small, but are very important in terms of developer productivity, code portability, and expressiveness. Any abstraction layer that would sit on top of both would discard the simplicity of one approach or the expressiveness of the other.

Peer-to-peer interoperability is much more attractive than a general abstraction layer. The xSDK libraries have mechanisms to work with or easily transform existing data objects that were built outside their own construction processes. For example, a PETSc sparse matrix can be used within Trilinos, without copying, by using an adapter class. A similar approach can work with a Trilinos matrix used by PETSc.

The hypre and SuperLU libraries do not directly support control inversion in the same way as PETSc and Trilinos, but do advertise their input data structures such that PETSc and Trilinos can construct compatible data structures that are passed to hypre and SuperLU without copying.

The current release of xSDK does not support all possible opportunities for interoperability. Level 1 interoperability is complete within the current xSDK. Level 2 interoperability is partial, with Trilinos being able to accept PETSc data structures. Level 3 interoperability is also partially available with PETSc and Trilinos able to call use hypre and SuperLU.

3.2. xSDK Community Policies

In [19, 20] various software quality engineering practices for 'smart libraries' are discussed that, when followed, can alleviate generation of an application executable that depends on many libraries, reduce mistakes in how to use these libraries, and provide help to users to identify and correct errors when they occur.

The first xSDK release demonstrated the impact of defining xSDK commmunity policies, including standard GNU autoconf and CMake options to simplify the combined use, portability, and sustainability of independently developed software packages (hypre, PETSc, SuperLU, and Trilinos) and provide a foundation for addressing broader issues in software interoperability and performance portability.

xSDK community package policies [22], briefly summarized in Figure 2, are a set of minimum requirements (including topics of configuring, installing, testing, MPI usage, portability, contact and version information, open source licensing, namespacing, and repository access) that a software package must satisfy in order to be considered xSDK compatible. The designation of xSDK compatibility informs potential users that a package can be easily used with others.

xSDK community installation policies [4] help make configuration and installation of xSDK software and other HPC packages as efficient as possible on common platforms, including standard Linux distributions and Mac OS X, as well as on target machines currently available at DOE computing facilities (ALCF, NERSC, and OLCF) and eventually on new exascale platforms.

xSDK Mandatory Policies

Must:

M1. Support xSDK community GNU Autoconf or CMake options [4].
M2. Provide a comprehensive test suite.
M3. Employ user-provided MPI communicator.
M4. Give best effort at portability to key architectures.
M5. Provide a documented, reliable way to contact the development team.
M6. Respect system resources and settings made by other previously called packages.
M7. Come with an open source license.
M8. Provide a runtime API to return the current version number of the software.
M9. Use a limited and well-defined symbol, macro, library, and include file name space.
M10. Provide an accessible repository (not necessarily publicly available).
M11. Have no hardwired print or IO statements.
M12. Allow installing, building, and linking against an outside copy of external software.
M13. Install headers and libraries under `<prefix>/include/` and `<prefix>/lib/`.
M14. Be buildable using 64 bit pointers. 32 bit is optional.

xSDK Recommended Policies

Should:

R1. Have a public repository.
R2. Possible to run test suite under valgrind in order to test for memory corruption issues.
R3. Adopt and document consistent system for error conditions/exceptions.
R4. Free all system resources it has acquired as soon as they are no longer needed.
R5. Provide a mechanism to export ordered list of library dependencies.

Figure 2. xSDK community policies specify expectations that any software library or framework (henceforth referred to as package) must satisfy in order to be xSDK compatible. The designation of a package being xSDK compatible informs potential users that the package can be easily used with other xSDK libraries and components and thus helps to address issues in long-term sustainability and interoperability among packages

Community policies for the xSDK promote long-term sustainability and interoperability among packages, as a foundation for supporting complex multiphysics and multiscale ECP applications. In addition, because new xSDK packages will follow the same standard, installation software and package managers (for example, Spack [10]) can easily be extended to install many packages automatically.

Figure 3 illustrates a new *Multiphysics Application C*, built from two complementary applications that can readily employ any libraries in the xSDK (hypre, PETSc, SuperLU, and Trilinos, shown in green). Application domain components are represented in orange. Of particular note is Alquimia [24], a domain-specific interface that support uniform access to multiple biogeochemistry capabilities, including PFLOTRAN [27]. The arrows among the xSDK libraries indicate current support for a package to call another to provide scalable linear solvers functionality on its behalf. For example, *Application A* could use PETSc for an implicit-explicit

time advance, which in turn could interface to SuperLU to solve the resulting linear systems with a sparse direct solver. *Application B* could use Trilinos to solve a nonlinear system, which in turn could interface to hypre to solve the resulting linear systems with algebraic multigrid. Of course, many other combinations of solver interoperability are also possible. The website `https://xsdk.info/example-usage` and [15] provide examples of xSDK usage, including interoperability among linear solvers in hypre, PETSc, SuperLU, and Trilinos.

3.3. xSDK Coordinated Software Releases and Current Status

The first critical step in producing the initial release of the xSDK in April 2016 was to define what the release should look like. On the loosely coupled end of the spectrum, one possibility would have been to certify that specific release versions of the different packages are compatible with one another and not coordinate the distribution of the release beyond that. On the other end of the spectrum were possibilities such as a common test or even build infrastructure. The xSDK team decided on a strategy that provided a single point of distribution for all xSDK component packages, but did not force a common infrastructure on the packages, beyond the agreed upon community policies.

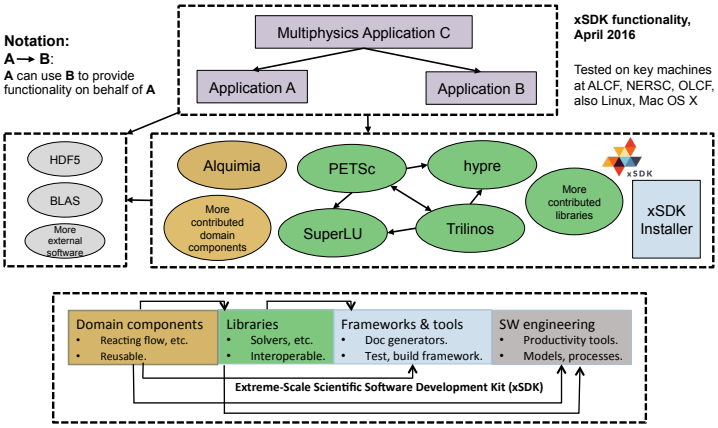


Figure 3. xSDK schematic diagram

The chosen distribution mechanism was, at its core, the existing PETSc distribution mechanism [3]. This was a stable, well-supported option that required a relatively small amount of effort to extend for the xSDK use case, because it already supported the majority of xSDK component packages, and very little additional ongoing maintenance beyond what the PETSc team was already doing. The latter was a key consideration because, when possible, the xSDK team actively avoids solutions that create long-term maintenance beyond what is needed for the component packages.

In addition to creating and updating interfaces between xSDK packages to provide new functionality, significant work was done to make it possible to install all of the xSDK packages together. For example, PETSc and Trilinos depended on different versions of SuperLU, and no single version of SuperLU could be used with both the PETSc and Trilinos SuperLU interfaces enabled.

As the coordinated release effort began (initially work began with package-to-package compatibility and interface efforts), the xSDK installer was used to test release versions of the component packages together to avoid the churn of the various development versions. Testing was set up by multiple xSDK component package teams using their own resources. Again, sustainability was a focus. Having the individual teams manage separate test builds in which each team had a vested interest was a better choice than a centralized effort that would have no clear owner in the absence of an xSDK-level funding source. The test builds included release

and development versions, but leading up to the release, primary focus was given to the release testing.

The existing release processes of the various xSDK component packages varied greatly in terms of testing and overall rigor. However, since each component package had a release process that the individual development teams had determined was sufficient for their needs, the decision was made to focus on requirements outside of typical release requirements. Specifically, this involved providing the name of a branch to use for release candidate testing, setting up tests for xSDK-level build configurations most relevant to the component package, and being responsive to any issues found.

In addition to the testing conducted by each package team, the xSDK 0.1 release was ported to three target platforms at three different computing facilities: Mira at ALCF, Edison at NERSC, and Titan at OLCF. One developer was primarily responsible for each of the three porting efforts, and those people coordinated with other xSDK developers and component package team developers to resolve porting issues as necessary.

The official tag for the initial xSDK release was chosen to be v0.1.0. After the initial release, a patch release, v0.1.1, was completed. The versions of the component packages used for this subsequent release were either the same version used for the initial 0.1 release, or a patch release of the component package based on the release used for the initial release. According to xSDK release policy, only patch-level updates for component packages are to be used for xSDK patch releases, and only patch or minor release version updates for component packages are to be used for xSDK minor releases.

Prior to selecting the PETSc distribution capability for the xSDK 0.1 release, Spack [10] was also seriously considered. Spack is a multi-platform package manager that supports a variety of compilers, libraries, and applications, as well as the installation of multiple concurrent versions and software configurations. Because of the increasing popularity and robustness of Spack, and the need to expand the xSDK to include several additional component packages, the xSDK team decided to use Spack to build an alpha release version of the xSDK, to be released in early 2017. Going forward, the intent is to use Spack as the principal supported xSDK distribution capability.

4. Next Steps

The first xSDK releases focused on discovery of collaboration models and community building among four of the major open source scientific library projects in the international scientific computing community: hypre, PETSc, SuperLU and Trilinos. The xSDK project will continue over the next few years under the United States Department of Energy Exascale Computing Project (ECP) [29].

Our efforts so far have established a baseline for expanding the xSDK scope under ECP funding in several important directions:

1. **Include more libraries:** The xSDK will expand to include all library efforts under ECP funding. Spcifically, widely used libraries such as SUNDIALS and Magma will become xSDK compatible, as will new efforts that address the performance challenges of exascale computing platforms.
2. **Further refine and expand community policies:** While the current xSDK community policies, summarized in Figure 2, are extremely useful as a mechanism to improve interoperability and compatibility of independently developed scientific libraries, we believe we can

further refine and expand these policies to better assure software quality and further realize the scientific software ecosystem sketched in Figure 1.

3. **Include more domain components:** As described in Section 1, the vision of the xSDK is to create a software ecosystem where new scientific applications are composed via interfaces from a common base of reusable domain components and libraries. We will work with science teams to identify opportunities for creating collections of domain components for their communities.
4. **Explore the use of community installation tools, including Spack:** While the extended PETSc installer has been very useful for establishing xSDK as a unified project, Spack [10] promises to provide a tool that serves and is supported by a larger community, making it very appealing as the principal long term installation tool for xSDK libraries.
5. **Process control transfer interfaces:** The ever-increasing use of concurrency within the top-level MPI processes requires that computational resources used by an application or library can be transferred to another library. Transfer of these resources is essential for obtaining good performance. The xSDK project will develop interfaces to support sharing and transfer of computational resources.

Conclusions

The extreme-scale scientific community faces numerous disruptive challenges in the coming decade. Fundamental limits of physics are forcing changes that dramatically impact all system layers from architecture to application software design. These disruptive changes drive us to move beyond incremental change in scientific application design and implementation. Establishing a scientific software ecosystem that focuses more on the composition of scalable, reusable components for application software development can provide an attractive alternative and the xSDK is the first step toward the ecosystem described in Figure 1.

Community policies for the xSDK promote long-term sustainability and interoperability among packages, as a foundation for supporting complex multiscale and multiphysics applications. The designation of xSDK compatibility informs potential users that a package can be easily used with other xSDK libraries and components. In addition, because new xSDK packages will follow the same standard, installxSDK and package managers can easily be extended to install many packages automatically.

Interoperability of xSDK member packages, when wrapped with adequate testing, enables a sustainable coupling of capabilities that enable applications to use xSDK member packages as a cohesive suite. The first xSDK releases demonstrate the impact of xSDK community policies, testing and examples to simplify the combined use, interoperability, and portability of independently developed software packages, establishing the first step toward realizing an extreme-scale scientific software ecosystem.

Acknowledgements

This material is based upon work funded by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs. We thank program managers Thomas Ndousse-Fetter, Paul Bayer, and David Lesmes for their support. The work of ANL authors is supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. The work of LBNL authors is partially supported

by the Director, Office of Science, Office of Advanced Scientific Computing Research of the US Department of Energy under contract no. DE-AC02-05CH11231.

Prepared by LLNL under Contract DE-AC52-07NA27344. Work of LANL authors is funded by the Department of Energy at Los Alamos National Laboratory under contract DE-AC52-06NA25396. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DEAC04-94AL85000.

References

1. Almgren, A., DeMar, P., Vetter, J., et al.: DOE Exascale Requirements Review for Advanced Scientific Computing Research (2016), u.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, see `http://exascaleage.org/ascr/`, Sept 27–29, 2016

2. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J.J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., Sorensen, D.: LAPACK Users' Guide (Third Ed.) (1999), DOI: 10.1137/1.9780898719604

3. Balay, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Rupp, K., Smith, B.F., Zhang, H.: PETSc Web page. `http://www.mcs.anl.gov/petsc`

4. Bartlett, R., Sarich, J., Smith, B., Gamblin, T., xSDK developers: xSDK community installation policies: GNU Autoconf and CMake options (2016), DOI: 10.6084/m9.figshare.4495133

5. Diachin (PI), L.: Interoperable Technologies for Advanced Petascale Simulations, SciDAC Institute. `http://www.scidac.gov/math/ITAPS.html`, accessed: 2017-02-15

6. Diachin (PI), L.: SciDAC-3 Frameworks, Algorithms, and Scalable Technologies for Mathematics (FASTMath) Institute. `http://www.fastmath-scidac.org/`, accessed: 2017-02-15

7. Dongarra, J.J., Bunch, J., Moler, C., Stewart, G.: LINPACK Users' Guide. SIAM Pub. (1979), DOI: 10.1137/1.9781611971811

8. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Software 16(1), 1–17 (March), DOI: 10.1145/77626.79170

9. Dongarra, J., DuCroz, J., Hammarling, S., Hanson, R.: An extended set of Fortran basic linear algebra subprograms. ACM Trans. Math. Software 14 (1988), DOI: 10.1145/42288.42291

10. Gamblin, T., LeGendre, M.P., Collette, M.R., Lee, G.L., Moody, A., de Supinski, B.R., Futral, W.S.: The Spack Package Manager: Bringing order to HPC software chaos. In: Supercomputing 2015 (SC'15). Austin, Texas (November 15-20), lLNL-CONF-669890

11. Garbow, B.S., Boyle, J.M., Dongarra, J.J., Moler, C.B.: Matrix Eigensystem Routines – EISPACK Guide Extension, Lecture Notes in Computer Science, vol. 51. Springer–Verlag, New York (1977), DOI: 10.1007/3-540-08254-9

12. Heroux, M.A., Allen, G., et al.: Computational Science and Engineering Software Sustainability and Productivity (CSESSP) Challenges Workshop Report (September 2016), https://www.nitrd.gov/PUBS/CSESSPWorkshopReport.pdf

13. Johansen, H., McInnes, L.C., Bernholdt, D., Carver, J., Heroux, M., Hornung, R., Jones, P., Lucas, B., Siegel, A., Ndousse-Fetter, T.: Software Productivity for Extreme-Scale Science (2014), report on DOE Workshop, January 13-14, 2014, available via http://www.orau.gov/swproductivity2014/SoftwareProductivityWorkshopReport2014.pdf

14. Keyes (PI), D.: Towards Optimal Petascale Simulations, SciDAC Institute. http://www.scidac.gov/math/TOPS.html, accessed: 2017-02-15

15. Klinvex, A.M.: xSDKTrilinos user manual. Tech. Rep. SAND2016-3396 O, Sandia (2016)

16. Lawson, C., Hanson, R., Kincaid, D., Krogh, F.: Algorithm 539: Basic linear algebra subprograms for Fortran usage. ACM Trans. Math. Software 5 (1979), DOI: 10.1145/355841.355848

17. Lawson, C., Hanson, R., Kincaid, D., Krogh, F.: Basic linear algebra subprograms for Fortran usage. ACM Trans. Math. Software 5 (1979), DOI: 10.1145/355841.355848

18. McInnes, L.C., Heroux, M., Li, X.S., Smith, B., Yang, with contributions from all xSDK developers, U.: What are Interoperable Software Libraries: Introducing the xSDK, version 0.1, April 25, 2016, available via https://ideas-productivity.org/resources/howtos/

19. Miller, M.C., Diachin, L., Balay, S., McInnes, L.C., Smith, B.: Package management practices essential for interoperability: Lessons learned and strategies developed for FASTMath (2013), DOI: 10.6084/m9.figshare.789055.v1

20. Miller, M.C., Reus, J.F., Matzke, R.P., Koziol, Q.A., Cheng, A.P.: Smart libraries: Best SQE practices for libraries with emphasis on scientific computing. In: Proceedings of the Nuclear Explosives Code Developer's Conference (2004), available via https://wci.llnl.gov/codes/smartlibs/UCRL-JRNL-208636.pdf

21. Rüde, U., Willcox, K., McInnes, L.C., De Sterck, H., Biros, G., Bungartz, H., Corones, J., Cramer, E., Crowley, J., Ghattas, O., Gunzburger, M., Hanke, M., Harrison, R., Heroux, M., Hesthaven, J., Jimack, P., Johnson, C., Jordan, K.E., Keyes, D.E., Krause, R., Kumar, V., Mayer, S., Meza, J., Mørken, K.M., Oden, J.T., Petzold, L., Raghavan, P., Shontz, S.M., Trefethen, A., Turner, P., Voevodin, V., Wohlmuth, B., Woodward, C.S.: Research and education in computational science and engineering (2016), available via https://arxiv.org/abs/1610.02608, submitted to *SIAM Review*

22. Smith, B., Bartlett, R., xSDK developers: xSDK community package policies (2016), version 0.3, December 2, 2016, DOI: 10.6084/m9.figshare.4495136

23. Smith, B.T., Boyle, J.M., Dongarra, J.J., Garbow, B.S., Ikebe, Y., Klema, V.C., Moler, C.B.: Matrix Eigensystem Routines – EISPACK Guide, Lecture Notes in Computer Science, vol. 6. Springer–Verlag, New York, second edn. (1976), DOI: 10.1007/3-540-07546-1

24. Alquimia Web page. https://bitbucket.org/berklab/alquimia, accessed: 2017-02-15

25. Hypre Web page. http://www.llnl.gov/CASC/hypre, accessed: 2017-02-15

26. NAG Web page. https://www.nag.com, accessed: 2017-02-15

27. PFLOTRAN Web page. http://www.pflotran.org, accessed: 2017-02-15

28. SuperLU Web page. http://crd.lbl.gov/~xiaoye/SuperLU, accessed: 2017-02-15

29. The Exascale Computing Project. https://exascaleproject.org, accessed: 2017-02-15

30. The HSL Mathematical Software Library. http://www.hsl.rl.ac.uk, accessed: 2017-02-15

31. Trilinos Web page. https://trilinos.org, accessed: 2017-02-15

DOI: 10.14529/jsfi170105

Performance Portability of HPC Discovery Science Software: Fusion Energy Turbulence Simulations at Extreme Scale

William Tang[1], *Bei Wang*[1], *Stephane Ethier*[2], *Zhihong Lin*[3]

As High Performance Computing Research and Development moves forward on a variety of "path to exascale" architectures today, an associated objective is to demonstrate performance portability of discovery-science-capable software. Important application domains, such as Magnetic Fusion Energy (MFE), have improved modeling of increasingly complex physical systems – especially with respect to reducing "time-to-solution" as well as "energy to solution." The emergence of new insights on confinement scaling in MFE systems has been aided significantly by efficient software capable of harnessing powerful supercomputers to carry out simulations with unprecedented resolution and temporal duration to address increasing problem sizes. Specifically, highly scalable particle-in-cell (PIC) programing methodology is used in this paper to demonstrate how modern scientific applications can achieve efficient architecture-dependent optimizations of performance scaling and code portability for path-to-exascale platforms. *Keywords: Turbulence Simulations, Particle-In-Cell, Portability, HPC.*

Introduction

A major challenge for supercomputing today is to demonstrate how advances in HPC technology translate to accelerated progress in key grand challenge application domains. This is the focus of an exciting new program in the US called the "National Strategic Computing Initiative (NSCI)" that was announced on July 29, 2015 involving all research & development (R&D) programs in the country to "enhance strategic advantage in HPC for security, competitiveness, and discovery." A strong focus in such prominent application domains is to accelerate progress in modern codes capable of modeling complex physical systems – especially with respect to reduction in "time-to-solution" as well as "energy to solution." In general, the demise of Dennard Scaling [1] coupled with the desire for processor and application performance to continue to track Moore's Law [10] has necessitated the switch from traditional superscalar processors to increasingly efficient processors built from lightweight cores and a hierarchical memory architecture. It is understood that if properly validated against experimental measurements/observational data and verified with mathematical tests and computational benchmarks, advanced codes can greatly improve much-needed predictive capability in many strategically important areas of interest.

As an illustrative example, computational advances in Magnetic Fusion Energy – a key scientific application area that was identified by the 2015 CNN "Moonshots for the 21st Century" series as one of five such prominent grand challenges – have produced particle-in-cell (PIC) simulations of turbulent kinetic dynamics for which computer run-time and problem size scale very well with the number of processors on massively parallel many-core supercomputers. For example, the GTC-Princeton (GTC-P) code, which has been developed with a "co-design" focus, has demonstrated the effective usage of the full power of current leadership class computational platforms worldwide at the petascale and beyond to produce efficient nonlinear PIC simulations that have advanced progress in understanding the complex nature of plasma turbulence and confinement in fusion systems for the largest problem sizes. Unlike fluid-like computational

[1]Princeton Institute for Computational Science and Engineering, Princeton University, Princeton, USA
[2]Princeton Plasma Physics Laboratory, Princeton, USA
[3]University of California Irvine, Irvine, USA

fluid dynamics (CFD) codes, PIC codes are characterized by having less than 10 key operations which can then be the focus of advanced computer science performance optimization methods. Results from these truly cross-disciplinary investigations have provided strong encouragement for being able to include increasingly realistic dynamics in extreme-scale computing campaigns with the goal of enabling predictive simulations characterized by unprecedented physics resolution/realism needed to help accelerate progress in delivering fusion energy.

1. Background

As the global energy economy makes the transition from fossil fuels toward cleaner alternatives, fusion becomes an attractive potential solution for satisfying the growing needs. Fusion energy, which is the power source for the sun, can be generated on earth, for example, in magnetically-confined laboratory plasma experiments (called "tokamaks") when the isotopes of hydrogen (e.g., deuterium and tritium) combine to produce an energetic helium "alpha" particle and a fast neutron, with an overall energy multiplication factor of 450:1. Building the scientific foundations needed to develop fusion power demands high-physics-fidelity predictive simulation capability for magnetically-confined fusion energy (MFE) plasmas. To do so in a timely way requires utilizing the power of modern supercomputers to simulate the complex dynamics governing MFE systems, including ITER, a multi-billion dollar international burning plasma experiment supported by 7 governments representing over half of the world's population. Currently under construction in France, ITER will be the world's largest tokamak system, a device that uses strong magnetic fields to contain the burning plasma in a doughnut-shaped vacuum vessel. In tokamaks, unavoidable variations in the plasma's ion temperature profile drive microturbulence, fluctuating electromagnetic fields, which can grow to levels that can significantly increase the transport rate of heat, particles, and momentum across the confining magnetic field. Since the balance between these energy losses and the self-heating rates of the actual fusion reactions will ultimately determine the size and cost of an actual fusion reactor, understanding and possibly controlling the underlying physical processes is key to achieving the efficiency needed to help ensure the practicality of future fusion reactors. The associated motivation drives the pursuit of sufficiently realistic calculations of turbulent transport that can only be achieved through advanced simulations. The present paper on advanced Particle-in-Cell (PIC) global simulations of plasma microturbulence at the extreme scale is accordingly associated with this fusion energy science (FES) grand challenge [11, 13].

Particle dynamics are well represented either by the 5D gyrokinetic (GK) equation (for low-frequency turbulence) or the 6D fully kinetic equation (for high frequency waves). The flagship GTC code [5] and it's "co-design" partner GTC-P [14, 15] are massively parallel particle-in-cell (PIC) codes designed to carry out first principles, integrated simulations of thermonuclear plasmas, including the future burning plasma International Thermonuclear Experimental Reactor (ITER). These codes solve the 5D GK equation in full, global toroidal geometry to address kinetic turbulence issues in magnetically-confined fusion experimental facilities.

GTC is the key production code for the fusion SciDAC Center "Gyrokinetic Simulation of Energetic Particle Turbulence and Transport (GSEP)" and for the Accelerated Application Readiness (CAAR) program at the Oak Ridge Leadership Computing Facility (OLCF). It is the only PIC code in the world fusion program capable of multiscale simulations of a variety of important physics processes in fusion-grade plasmas including microturbulence, energetic particle dynamics, collisional (neoclassical) transport, kinetic magnetohydrodynamic (MHD) modes,

and nonlinear radio-frequency (RF) waves. GTC interfaces with MHD equilibrium solvers for addressing realistic toroidal geometry features that include both axisymmetric tokamaks and non-axisymmetric stellarators. A recent upgrade enables this code to carry out global PIC simulations covering both the tokamak core and scrape-off layer (SOL) regions. It should also be noted that the current comprehensive version of GTC can carry out both perturbative (δ f) and non-perturbative (full-f) simulations with capability of dealing with kinetic electrons, electromagnetic fluctuations, multiple ion species, collisional (neoclassical) effects using Fokker-Planck collision operators, equilibrium current and radial electric field, plasma rotation, sources/sinks, and external antennae for auxiliary wave heating. Beyond the conventional application domain of gyrokinetic simulation of microturbulence, the GTC code has a long history in pioneering the development and application of gyrokinetic simulations of meso-scale electromagnetic Alfven eigenmodes excited by energetic particles (EP) in toroidal geometry. This is one of the most important scientific challenges that must be addressed in future burning plasma experiments such as ITER. Accordingly, the GTC work-scope has recently been extended to include simulation of macroscopic kinetic-MHD modes driven by equilibrium currents. The associated importance is that such efforts could ultimately lead to key knowledge needed to systematically analyze and possibly help avoid or mitigate highly dangerous reactor relevant thermonuclear disruptions.

The GTC-P code is a performance-optimized, highly portable modern PIC code that serves as a "co-design" proxy for the flagship GTC code. It serves to help accelerate progress on architecture-dependent optimization including scalability and portability for emerging exascale computers with heterogeneous architectures including both the GPU-accelerated Summit available in 2018 at ORNL and many-core system Aurora available in 2019 at ANL. The associated focus of GTC-P involves evaluation and implementation into GTC the emerging standards-based programming models that may enable performance portability across many-core and GPU-accelerated architectures.

The current transition from traditional superscalar processors to increasingly energy-efficient processors built from lightweight cores and a hierarchical memory architecture can be expected to be a trend that will persist into the next 5 years. Many-core processors (Intel KNL/KNH) and NVIDIA's GPU-accelerators provide a management technology that frees end users from needing to micromanage data movement and data locality. Over the past decade, strong collaborative interactions between Princeton and LBNL have delivered increasingly improved versions of GTC-P on some of the fastest supercomputers in the world including a series of GPU-accelerated systems and the first generation of Intel many-core coprocessors – culminating in the recent publication [14] that also involved key contributions from ETH-Zurich. The increased prominence of highly threaded architectures coupled with the desire to minimize memory usage has led to the need to implement novel domain decomposition, synchronization, and particle binning techniques. For example, the deployment of domain decomposition in the radial dimension has been demonstrated in GTC-P [15] to dramatically reduce the memory footprint and the associated computational work for grid-based subroutines. This unique capability, which has not been implemented in most fusion codes, is targeted for implementation into GTC. More generally, the novel techniques developed in GTC-P including the use of floating-point atomics on GPUs, as well as the pragmatic approaches required for efficient vectorization on the Knights family will also be integrated into GTC. We will plan to continue our exploration of pipelined, one-sided communication (MPI or UPC++) in order to efficiently and productively implement electron particle pushing and shifting. GTC-P is expected to have an increasingly enhanced role as the

GTC co-design proxy for interaction with both the computational centers (OLCF at ORNL, ALCF at ANL, and NERSC at LBNL) and vendors (Intel, NVDIA, IBM), who can tweak their respective offerings for the computational requirements of GTC in the pre-exascale and exascale timeframes. This will all help achieve the goal of efficiently carrying out architecture-dependent optimization of GTC kernels to help ensure performance portability across both large many-core and GPU systems.

2. Scientific Methodology

The GTC and GTC-P codes include all of the important physics and geometric features captured in numerous global PIC simulation studies of plasma size scaling over the years, extending from the seminal work in the Phys. Rev. Letter (PRL) by Z. Lin, et al. [4] up to the more recent PRL paper by B. F. McMillan, et al. [9] on system size effects on gyrokinetic turbulence. The current generally supported picture is that size-scaling follows an evolution from a "Bohm-like" trend where the confinement degrades with increasing system size, to a "Gyro-Bohm-like" trend where the confinement for JET-sized plasmas begins to "plateau" and then exhibits no further confinement degradation as the system size further increases toward ITER-sized plasmas. A number of physics papers over the past decade have proposed theories, such as turbulence spreading, to account for this transition to Gyro-Bohm scaling with plasma size for large systems. From a physics perspective, this key decade-long fusion physics picture of the transition or "rollover" trend associated with toroidal ion temperature gradient micro-instabilities that are highly prevalent in tokamak systems, should be re-examined by modern supercomputing-enabled simulation studies which are now capable of being carried out with much higher phase-space resolution and duration. With a focused approach based on performance optimization of key functions within PIC codes in general, GTC-P, the "co-design" focus, has demonstrated the effective usage of the full power of current leadership class computational platforms worldwide at the petascale and beyond to produce efficient nonlinear PIC simulations that have advanced progress in understanding the complex nature of plasma turbulence and confinement in fusion systems for the largest problem sizes. Unlike fluid-like computational fluid dynamics (CFD) codes, GTC-P has concentrated on the fact that PIC codes are characterized by having less than 10 key operations which can then be an especially tractable target for advanced computer science performance optimization methods. As illustrated in [14], these efforts have resulted in accelerated progress in a discovery-science-capable global PIC code that models complex physical systems with unprecedented resolution and produces valuable new insights into reduction in "time-to-solution" as well as "energy to solution" on a large variety of leading supercomputing systems. In a sense, GTC-P is a "co-desing proxy" for the "flagship" electromagnetic GTC code which is the most comprehensive PIC code with respect to the complex physics included. GTC has delivered many scientific advances while using increasingly powerful supercomputing systems over the years. For example, it is the first large-scale fusion code to deliver production run stimulations at the terascale in 2002 [4] and on a petaflop system in 2009 [17]. Several key associated computational methodologies will be elaborated upon in the subsequent sections of this paper.

2.1. Global PIC Geometric Models

In plasma turbulence studies, the standard approach is to divide the physical quantities into an equilibrium part and a fluctuating part. The GTC code uses two set of meshes, one for the specification of the equilibrium and the other to represent fluctuating turbulent fields. In particular, the turbulence mesh is an unstructured field-aligned mesh for finite difference or finite element in 3D space.

The equilibrium quantities are governed by the Grad-Shafranov equation for toroidal geometry, while the fluctuating part is driven by various instabilities that lead to turbulent transport. Equilibrium magnetic configurations typically used in gyrokinetic simulations either come from: (i) analytic models such as the simple circular cross section or the Miller equilibrium; and (ii) numerical equilibrium codes such as EFIT or VMEC. For the rapidly evolving optimization studies that deliver very high resolution results from investigations of plasmas with increasing problem size on the most powerful supercomputing systems, the practical choice, as exemplified by the "co-design" GTC-P code – is the category (i) analytically-based equilibria. On the other hand, comprehensive production runs carried out by the flagship GTC code demand interfacing with the numerical equilibria of category (ii) that properly represent the actual experimental conditions.

The most accurate representation of the equilibrium in tokamaks is by using magnetic flux coordinates rather than Cartesian coordinates. This is due to the fact that most important equilibrium quantities, such as plasma temperature and density, can be shown to depend on the magnetic flux only. The flagship GTC code employs magnetic flux coordinates (Ψ, θ, ζ) to represent the electromagnetic fields and plasma profiles, where Ψ is the poloidal flux function, θ is the poloidal angle, and ζ is the toroidal angle. Specifically, the inputs come from the numerical magnetic equilibrium and plasma profiles obtained from EFIT/VMEC by transforming the equilibrium quantities defined in the cylindrical coordinates (R, ϕ, Z) to those defined in the magnetic coordinates (Ψ, θ, ζ). The equilibrium data are provided by MHD equilibrium codes for the magnetic field strength B, and cylindrical coordinates (R, ϕ, Z) of points forming magnetic flux surfaces. Additionally, the flux functions representing poloidal $g(\Psi)$ and toroidal $I(\Psi)$ currents, magnetic safety factor $q(\Psi)$, and minor radius $r(\Psi)$ – defined as a distance from the magnetic axis along the outer mid-plane – are provided. First-order continuous B-splines are implemented for the 1D, 2D, and 3D functions to interpolate the complicated magnetic geometry and plasma profiles which provide a good compromise between high numerical confidence and reasonable computational efficiency.

The GTC capability to carry out simulations of problems with general toroidal geometry has recently been extended to also include non-axisymmetric configurations. For non-axisymmetric devices, the equilibrium data are presented on the uniform (Ψ, θ) grid for all n=(1, 2, $\cdots$, N) toroidal harmonics. To reduce the computational load and memory usage, the transformation of non-axisymmetric variables into spline functions of ζ is chosen for implementation in GTC, with spline coefficients associated with a particular grid point ζ_i being stored by processors with corresponding toroidal rank using message passing interface (MPI) parallelization.

The GTC-P code deploys the so-called large aspect ratio equilibrium, which is an analytical model describing a simplified toroidal magnetic field with a circular cross-section. The associated model takes into account the key geometric and physics properties needed to carry out a meaningful study of the influence of increasing plasma size on magnetically-confined fusion plasmas. Such an approach enables working with a sufficiently straightforward but nevertheless

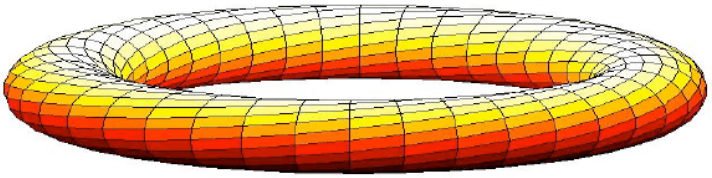


Figure 1. Illustrative Figure showing the grid structure of the GTC-P code on a 3D torus

discovery-science-capable physics [4,5] code that makes more tractable the formidable task of developing the algorithmic advances needed to take advantage of the rapidly evolving modern platforms featuring, for example, both homogenous and hybrid architectures. The associated physics approach is to deploy GTC-P plasma size-scaling studies because it is a fast streamlined modern code with the capability to efficiently carry out computations at extreme scales with unprecedented resolution and speed on present-day multi-petaflop computers [14]. The corresponding scientific goal is to accelerate progress toward capturing new physics insights into the key question of how turbulent transport and associated confinement characteristics scale from present generation laboratory plasmas to the much larger ITER-scale burning plasmas. This includes a systematic characterization of the spectral properties of the turbulent plasma as the confinement scaling evolves from a "Bohm-like" trend where the confinement degrades with increasing system size to a "Gyro-Bohm-like" trend where the confinement basically "plateaus", exhibiting no further confinement degradation as the system size further increases. "Lessons learned" achieved in a timely way from this co-design effort can be expected to expedite associated advances in the flagship GTC code in particular as well as to generally providing valuable information on PIC performance modeling advances to ongoing and future efforts in improving PIC code deployment on multi-petaflop supercomputers on the path to exascale and beyond.

2.2. Global PIC Grid Considerations

To accurately track the key physics in magnetically-confined toroidal plasmas, the GTC and GTC-P codes utilize a highly specialized grid that follows the magnetic field lines as they twist around the torus (see Figure 1). This allows the code to retain the same accuracy while using fewer toroidal planes than a regular, non-field-aligned grid. From relevant physics considerations, since short wavelength waves parallel to the magnetic field are suppressed by Landau damping, increasing the grid resolution in the toroidal dimension will leave the results essentially unchanged. Consequently, a typical production simulation run usually consists of a constant number of poloidal planes (e.g., 32 or 64) wrapped around the torus. Each poloidal plane is represented by an unstructured grid, where the grid sizes in the radial and poloidal dimensions correspond approximately to the size of the gyro-radius of the particles. As we consider larger plasma sizes (e.g., 2x in major and minor radius), the number of grid points in each 2D plane increases 4x. The number of grid points for a 3D grid increases 4x as well since the number of planes in the toroidal dimension remains the same for all problem sizes. For a modest-sized fusion device (e.g, the DIII-D tokamak at General Atomics in San Diego, CA), the associated plasma simulation typically uses ∼128 thousand grid points in a 2D plane. As we move to the larger Joint European Torus (JET) device and then eventually to the ITER size plasmas, the number of grid points increases 4x and 16x, respectively. Using a fixed number of 64 toroidal planes, the total number of grid points for an ITER-sized plasma will be ∼131 million. With

100 particles per cell resolution, an ITER-sized simulation will accordingly involve ∼13 billion particles. For E&M and full-f simulation with 3 particle species, the particle number can even increase to 10^{11}. Tracking the dynamics of this large number of particles would of course be an extremely daunting task without access to leadership-class supercomputers.

3. Programming Approach

The basic parallel programming approach for global PIC codes such as GTC and GTC-P includes: (i) explicit message passing using MPI; (ii) architecture-specific models such as CUDA for computing on GPUs; and (iii) directive-based compiler options such as OpenMP and OpenACC with possible promise of being more cross-machine portable between architectures.

A more detailed recent description of global PIC code characteristics/considerations with respect to scalability, performance, portability, modern computational platforms, and external libraries, associated discussions will touch on the rationale for the chosen programming approach, and the associated balance between performance and portability can be found in [14]. In future R&D, attention must of course be focused on the many specific challenges for global PIC applications in achieving efficiency on exascale architectures. A preview is given in the following sections.

3.1. PIC Scalability

In describing efforts to improve the performance scalability of the global PIC codes – well represented by GTC and GTC-P, key topics include: (i) on-node thread scaling; and (ii) between node scaling. The GTC/GTC-P codes have been designed with four levels of parallelism: (i) an inter-node distributed memory domain decomposition via MPI, (ii) an inter-node distributed memory particle decomposition via MPI, (iii) an intra-node shared memory work partition implemented with OpenMP; and (iv) a SIMD vectorization within each core. This approach was shown to lead to nearly-perfect scaling with respect to the number of particles [2].

In order to efficiently address large grid sizes and the associated significant memory increase, the domain decomposition in GTC-P is further extended in the radial dimension (beyond the toroidal dimension) [15]. This leads to a 2D domain decomposition and enables carrying out true weak scaling studies, where both particle and grid work are appropriately scaled. The multi-level particle and domain decompositions provide significant flexibility in distributed-memory task creation and layout. While the ranks in the toroidal dimension are usually fixed as 32 or 64 due to Landau damping physics, there is freedom to choose any combination of process partitioning along the radial and particle dimensions. For scaling with a fixed problem size, the procedure involves partitioning along the radial direction and using particle decomposition. The decompositions were implemented with three individual communicators in MPI (toroidal, radial, and particle communicator), and further tuning is made available via options to change the order of MPI rank placement.

It is important to note that gyrokinetic PIC simulations typically exhibit highly anisotropic behavior – with the velocity parallel to the magnetic field being an order of magnitude larger than that in the perpendicular direction. Consequently, the message sizes in the toroidal dimension can be many times larger (e.g., an order) than those in the radial dimension at each time step. On the IBM Blue Gene systems with explicit process mapping, it was found to be convenient and effective to group processes to favor the MPI communicator in the toroidal dimension. For other

systems, assigning consecutive ranks for processes within each toroidal communicator generally leads to improved performance.

Looking toward the ongoing and future challenge of maximizing on-node performance and efficiency, it is already clear that modern processor architectures have evolved with more cores and wider vector units in a single node. In order to fully exploit the emerging architectures on the path to exascale, it is important that application scientists design their software such that the algorithms and the implementations map well on the hardware for maximum scalability. In GTC/GTC-P, this translates to multicore parallelism using shared-memory multi-threading and implementation changes to enable SIMD vectorization. For example, in an earlier version of GTC-P, "holes" were used to represent non-physical "invalid" particles, i.e., in a distributed environment, at every time step, the particles that are being moved to other processes are marked as "holes" and considered to be "invalid" in the local particle array. These invalid particles are then removed from the array periodically to empty memory space for new incoming particles. In this type of implementation, two particles in consecutive memory locations may have different operations in charge and push depending on if they belong to the same type of particles (valid or invalid) or not. This accordingly introduces difficulty for automatic vectorization. To maximize the usage of vector units, the latest version of GTC-P includes removing the holes completely for charge and push by filling the holes at the end of shift and using the new incoming particles sent from neighboring processors at every time step. If a process has sent more particles than received, then the remaining holes are filled with the last particles in the array. A similar strategy has been applied for the GPU implementation to remove the branch statement caused by the "holes".

3.2. PIC Performance Challenges

PIC algorithms are challenging to optimize on modern computer architectures due to issues such as data conflict and data locality. In GTC and GTC-P, parallel binning algorithms have been developed to improve data locality for charge and push. More specifically, several choices are provided to bin the particles, i.e., along the radial dimension and along the poloidal dimension. The best binning strategy will be used for production runs by first running a few benchmarks. In GTC-P, the additional use of intrinsics has helped improve the vectorization of the binning implementation. On GPU's, the CUDA version of the binning algorithm was implemented using the Thrust Library.

To address the data conflict issue in charge, optimization strategies have been explored via static replication of grid segments that are coupled with synchronization via atomics, where the size of the replica may be traded for increased performance [7, 8]. The best performance is often obtained by employing the full poloidal grid for each OpenMP thread. In GTC with only toroidal domain decomposition, the full poloidal grid replication dramatically increases the temporary grid-related storage for large size grid on manycore architectures such as the Intel Xeon Phi systems. As such, static replication of grid segments that are coupled with synchronization via atomics will likely be the best strategy. In GTC-P, the radial domain decomposition solves locality and memory pressure without resorting to costly atomics. In essence, since only a small segment of the full poloidal grid is required for a hazard-free charge deposition, the private grid replication strategy can be readily employed on a per thread basis for the best performance.

In dealing with heterogenous supercomputing platforms such as "Titan", the approach followed in the deployment of global PIC codes involves off-loading the computationally intensive

and highly scalable subroutines to GPU's, while the communication-dominant subroutines remain on CPU's [3]. Performance, however, is known to be impeded due to the synchronization of atomic operations and the unavoidable memory transpose associated with the structure-of-array to array-of-structure data layout. To address this issue, the time-consuming global memory atomic operations have been replaced with local shared memory atomic operation. This R&D activity falls generally in the category of advances and challenges involving heterogeneous architectures.

3.3. Portability

Global PIC codes such as GTC and GTC-P have demonstrated increasing capability for portability over the past few years across different architectures. In this section the associated techniques applied for doing so are discussed along with examples of success achieved. In general, a high priority is being placed on portability in HPC because of the significant differences between quite different architectural approaches such as the upcoming 200 PF systems: SUMMIT at the OLCF and AURORA at the ALCF. Since both approaches have significant exciting potential for enabling accelerated performance at scale, most advanced applications, including prominent global PIC codes such as GTC/GTC-P, will continue to focus attention on achieving both performance enhancement as well as portability. For example, performance portability of these advanced codes helps ensure, in a risk mitigation sense, the capability to perform very well on whichever platform proves to provide the greater eventual computing at extreme scale advantage.

GTC-P has been particularly successful in porting modern optimized versions across a wide range of multi peta-flop platforms at full or near-to-full capability. Benefit is associated in part from the fact that GTC-P is not critically dependent on any third-party libraries. This includes the implementation a highly-optimized Poisson solver with multi-threading capability. Additional performance enhancement for both GTC and GTC-P has been obtained by utilizing a specialized damped Jacobi iterative solver [6]. In this iterative solver, the damping parameter was carefully chosen to favor the desired range of wavelengths for the fastest growing modes in plasma turbulence simulations. As a result, a small and fixed iteration count is sufficient to achieve the desired accuracy.

Although achieving the best performance on each explored architecture requires platform-specific optimization strategies, a "pluggable" software component approach in architecting the GTC/GTC-P application codes have proven to be a quite successful approach. Specifically, the interface is preserved across all implementations targeting CPU-based codes as well as GPU (or Xeon Phi) hybrid implementations. Components are chosen based on the target platform during the application build process. This enables having a unified code base with the best-possible performance, without sacrificing portability. Behind the unified interface, platform-specific optimization strategies are systematically investigated.

Some optimizations, such as sorting particles and vectorization, are common to all platforms, but implementation details differ. Other optimizations, such as handling NUMA issues and load imbalance, are specific to certain platforms. Designing routine interfaces is of crucial importance to allow portability without compromising performance-tuning opportunities.

GTC-P uses the MPI-3 standard for distributed-memory communication, including the exploration of one-sided communication. The motivation here is again to provide better portability for diverse architectures and programming models.

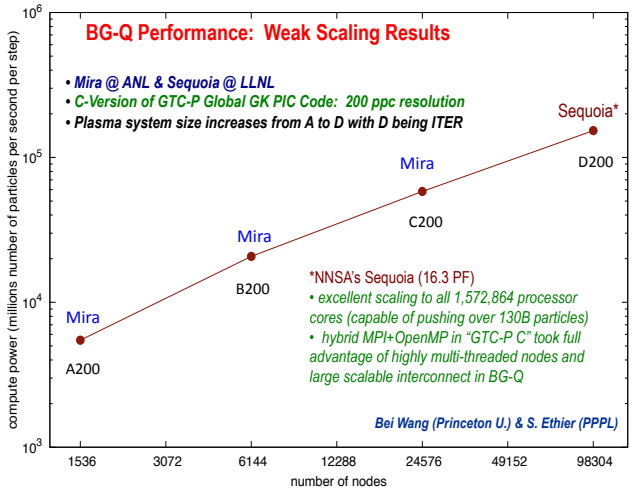


Figure 2. GTC-P Code Performance on World-Class IBM BG/Q Systems

Significant advances in GTC-P on many-core processors with respect to portability and scalability have been recently achieved by porting the code to GPU systems with OpenACC 2.0 as a viable option instead of CUDA. This has led to the very recent success in porting and optimizing an OpenACC 2.0 version of GTC-P on the Sunway TaihuLight Supercomputer [16] – the new No. 1 system on the international Top500 as of June 2016. The only approach for achieving good performance on TaihuLight requires software compatibility with their SWACC compiler, a customized OpenACC 2.0 syntax supported software.

In common with the large majority of codes in the fusion energy science/plasma physics application domain, GTC-P was originally written in Fortran language. However, to better facilitate interdisciplinary collaborations with computer science and applied math colleagues, modern versions of this code have been developed in C language as well as a CUDA implementation for dealing with GPU's. As just noted, this capability has recently been further advanced with the development and implementation of an OpenACC 2.0 version of GTC-P. Although the original Fortran version of this code is still used for verification purposes in cross-checking and benchmarking results, the primary utilization has involved the C and CUDA versions for performance studies and physics production runs on supercomputing systems such as the ALCF's "Mira" and the OLCF's "Titan".

4. Scaling Results

Using resources from INCITE, previous early Science Projects (ESP) at the ALCF, and Director's Discretionary allocations from both the ALCF and OLCF in the past few years, GTC-P has demonstrated excellent scalability to more than 100,000 cores on leadership computing facilities at ANL and ORNL. It has been successfully deployed for major scientific production runs on the IBM BG/Q/"Mira", where the excellent weak scaling performance was carried over to much larger scale on LLNL's more powerful Sequoia system. These results are illustrated in Figure 2.

In addition, it is relevant to note that the GTC-P code was the featured U.S. code in the G8 international exascale project in nuclear fusion energy, "NuFuSE" that was supported in

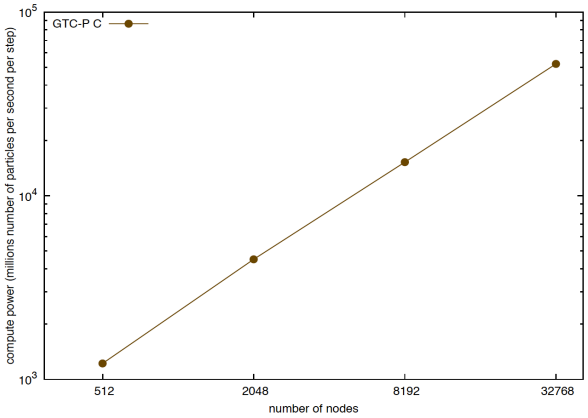


Figure 3. Weak Scaling of the GTC-P code Achieved on the Fujitsu-K Computer in Japan

the U.S. by the National Science Foundation (NSF). The G8 program helped provide unique access to a variety of international leadership class computational facilities such as the Fujitsu K Computer in Japan. Results from weak-scaling studies carried out on the K-computer are illustrated in Figure 3.

As seen from subsequent stimulating new results [12], substantive impact can be expected to help stimulate progress in preparing for actual research engagement on ITER. In order to do so in a timely way, it is critically important that new software for extreme concurrency systems that demand increasing data locality be developed to help accelerate progress toward the ultimate goal of computational fusion research, a predictive simulation capability that is properly validated against experiments in regimes relevant for practical fusion energy production.

Having demonstrated the ability to effectively utilize the most powerful homogeneous supercomputing platforms worldwide, GTC-P R&D efforts have also examined performance characteristic on heterogeneous architectures. More generally, these studies represent productive investigations of extreme scale science across advanced scientific computing basic research programs with fusion energy science as an illustrative application domain. Here the focus was on developing new algorithms for advanced heterogeneous supercomputing systems such as the GPU/CPU "Titan" at DOE's OLCF and the Intel Xeon Phi/Intel Xeon "Stampede" at NSF's TACC. In doing so, a new version of GTC-P code was developed that features algorithms which include new heterogeneous capabilities for deployment on hybrid GPU (Nvidia K20)/CPU as well as the Intel Xeon Phi/Intel Xeon systems such as Stampede and also TH-2 in China. From a verification perspective, this research effort also includes systematic comparison of new results against the successful work described earlier in studies that featured high resolution, long temporal scale simulation results obtained on world-class homogeneous systems such as the IBM BG/Q Mira at the ALCF, Sequoia at LLNL, and the K-Computer in Kobe, Japan. A weak scaling performance of GTC-P across a wide-range of systems is shown in Figure 4. Interested readers can also find more details in [14].

The CAAR program has enabled GTC architecture-dependent optimization including scalability and portability for heterogeneous architectures. Figure 5 has shown the GTC hybrid weak scaling study (i.e., same number of particles but increasing number of grid points per process) where we scale both the grid size and the total number of particles from DIIID to ITER simulations on Titan up to 16384 nodes [18]. Compared with CPU (16 cores AMD 6274), GPU (NVIDIA K20x) has boosted the overall performance by 1.5-2.8x. The decrease of the perfor-

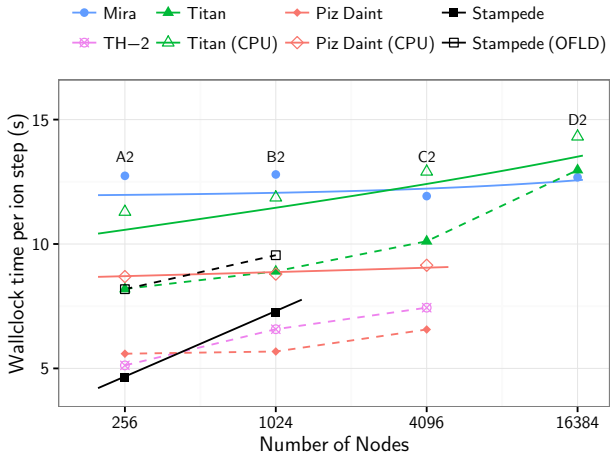


Figure 4. GTC-P weak scaling performance using a fixed problem size per node across all systems allows comparisons of node performance. Solid lines indicate model-predicted running times (shown for Mira, Titan (CPU), Piz Daint (CPU), Stampede), dashed line joins actual running times

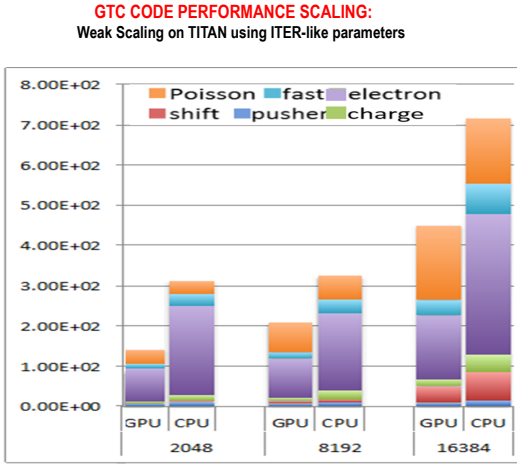


Figure 5. GTC hybrid weak scaling study where we scale both the grid size and total number of particles from DIIID to ITER simulations on Titan up to 16384 nodes

mance speed up in large processor counts is due to the increased portion of non-GPU accelerated subroutines.

5. Time-to-Solution and Energy-to-Solution Comparative Studies

As evident from the increasingly strong attention paid to "Green 500" in addition to the traditional "Top 500" supercomputer rankings, energy is clearly being recognized as a prominent impediment to advances in HPC development. Since the interplay between performance and power is highly dependent on algorithm and architecture, assessing the net energy efficiency of large scientific simulations can be particularly non-intuitive as one moves from one processor or network architecture to the next. In Table 1 (also shown in Table V of [14]), the energy per time step was illustrated for 4K nodes of Mira, Titan, and Piz Daint when using 80M

Table 1. Energy per ion time step (KWh) by platform for the weak-scaled, kinetic electron configuration at 4096 nodes. Power is obtained via system instrumentation including compute node, network, blades, AC to DC conversion

	CPU-Only			CPU+GPU	
	Mira	**Titan**	**Piz Daint**	**Titan**	**Piz Daint**
Nodes	4096	4096	4096	4096	4096
Power/node (W)	69.7	254.1	204.9	269.4	246.5
Time/step (s)	13.77	15.46	10	10.11	**6.56**
Energy (KWh)	**1.09**	4.47	2.33	3.10	1.84

grid points, 8B ions, and 8B electrons. The power measured under actual load via system instrumentation was used. Attention should be paid to the fact that although Mira required the most wall clock time per time step, it also required the least power per node. Taken as whole, the conclusion was that Mira required the least energy per time step of all platforms considered. Conversely, using the host-only configurations on Titan and Piz Daint required between 2x and 4x the energy with the difference largely attributable to the relative lack of scalability on Titan. It is especially interesting to highlight that while code acceleration on these platforms significantly reduced the wall clock time per time step, the associated power expended was only slightly increased. Consequently, the energy required for the GPU-accelerated systems was reduced nearly proportionally with run time.

With regard to energy-efficient scientific computing, instrumenting scientific applications to measure energy when running on large supercomputing installations today can be cumbersome and obtrusive – requiring significant interaction with experts at each center. As such, most applications have little or no information on energy-to-solution across the architecture design space spectrum. In order to affect energy-efficient co-design of supercomputers, energy measurement must always-on by default with, at a minimum, total energy and average power reported to the user at the end of an application. By reporting energy by component (memory, processor, network, storage, etc...), scientists and vendors could co-design their applications and systems to avoid energy hotspots and produce extremely energy-efficient computing systems.

Conclusions

Portability across many-core and GPU-accelerated architectures is important to the success of this illustrative PIC GTC framework highlighted in this paper. This discovery science software will have a large user community and needs to efficiently utilize all computational resources of the next generation computers. Further studies will continue to explore the efficacy of using the OpenMP 4.5 offload model and the use of OpenACC to provide performance portability across architectures in lieu of using both CPU-specific OpenMP implementations and GPU-specific OpenACC implementations. We note that the performance portability of OpenACC has not been studied on Intel's KNL/KNH processors and the OpenMP offload model is not fully supported on many compilers. Moreover, given the offload model has been rendered irrelevant by KNL's self-hosted nature coupled with its use of MCDRAM as a hardware-managed cache, there is a distinct possibility that standards-based single-source performance portability will prove elusive and some specialization will remain essential. We will continue to leverage GTC-P as GTC's co-design proxy for interaction with both the computational centers (OLCF at ORNL, ALCF at ANL, and NERSC at LBNL) and vendors (Intel, NVDIA, IBM), who can tweak their

respective offerings for the computational requirements of GTC in the pre-exascale and exascale timeframes.

As a final comment, it is appropriate to note that a broader impact of the work presented in this paper is the delivery of benefits to particle-in-cell codes in general since the associated codes share a common algorithmic foundation. For example, the continuing developments targeted in the global PIC GTC project can be expected to have a strong impact in dealing with the multi-threading challenges for efficient deployment of a large number of processors on modern homogeneous and heterogeneous systems advances that should prove beneficial to any particle-mesh algorithm.

Acknowledgements

The authors are very grateful to our many collaborative colleagues. We are especially indebted to Sam Williams and Khaled Ibrahim from LBNL.

References

1. Dennard, R.H., Gaensslen, F.H., Rideout, V.L., Bassous, E., LeBlanc, A.R.: Design of ion-implanted mosfet's with very small physical dimensions. IEEE Journal of Solid-State Circuits 9(5), 256–268 (Oct 1974), DOI: 10.1109/jssc.1974.1050511

2. Ethier, S., Tang, W.M., Lin, Z.: Gyrokinetic particle-in-cell simulations of plasma microturbulence on advanced computing platforms. Journal of Physics: Conference Series 16, 1–15 (2005), DOI: 10.2172/1222712

3. Ibrahim, K.Z., Madduri, K., Williams, S., Wang, B., Ethier, S., Oliker, L.: Analysis and optimization of gyrokinetic toroidal simulations on homogenous and heterogenous platforms. International Journal of High Performance Computing Applications (2013), DOI: 10.1145/2063384.2063415

4. Lin, Z., Ethier, S., Hahm, T.S., Tang, W.M.: Size scaling of turbulent transport in magnetically confined plasmas. Phys. Rev. Lett. 88, 195004 (Apr 2002), DOI: 10.1145/1654059.1654108

5. Lin, Z., Hahm, T.S., Lee, W.W., Tang, W.M., White, R.B.: Turbulent transport reduction by zonal flows: Massively parallel simulations. Science 281(5384), 1835–1837 (1998), DOI: 10.1126/science.281.5384.1835

6. Lin, Z., Lee, W.W.: Method for solving the gyrokinetic poisson equation in general geometry. Phys. Rev. E 52, 5646–5652 (Nov 1995), DOI: 10.1145/2063384.2063415

7. Madduri, K., Ibrahim, K.Z., Williams, S., Im, E.J., Ethier, S., Shalf, J., Oliker, L.: Gyrokinetic toroidal simulations on leading multi- and manycore HPC systems. In: Proc. Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '11). pp. 23:1–23:12. ACM, New York, NY, USA (2011), DOI: 10.2172/1273408

8. Madduri, K., Williams, S., Ethier, S., Oliker, L., Shalf, J., Strohmaier, E., Yelick, K.: Memory-efficient optimization of gyrokinetic particle-to-grid interpolation for multicore processors. In: Proc. ACM/IEEE Conf. on Supercomputing (SC 2009). pp. 48:1–48:12 (Nov 2009), DOI: 10.1145/2616498.2616526

9. McMillan, B.F., Lapillonne, X., Brunner, S., Villard, L., Jolliet, S., Bottino, A., Görler, T., Jenko, F.: System size effects on gyrokinetic turbulence. Phys. Rev. Lett. 105, 155001 (Oct 2010), DOI: 10.1103/physrevlett.105.155001

10. Moore, G.E.: Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. IEEE Solid-State Circuits Society Newsletter 11(5), 33–35 (Sept 2006), DOI: 10.1088/1742-6596/16/1/001

11. Rosner, R., et al.: Opportunities and challenges of exascale computing - doe advanced scientific computing advisory committee report (2010), `https://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf`, accessed: 2017-02-15

12. Tang, W., Wang, B., Ethier, S.: Scientific discovery in fusion plasma turbulence simulations at extreme scale. Computing in Science Engineering 16(5), 44–52 (Sept 2014), DOI: 10.1103/physrevlett.103.085004

13. Tang, W., Keyes, D.: Scientific grand challenges: Fusion energy science and the role of computing at the extreme scale. In: PNNL-19404. p. 212 (2009), DOI: 10.1177/1094342013492446

14. Tang, W., Wang, B., Ethier, S., Kwasniewski, G., Hoefler, T., Ibrahim, K.Z., Madduri, K., Williams, S., Oliker, L., Rosales-Fernandez, C., Williams, T.: Extreme scale plasma turbulence simulations on top supercomputers worldwide. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 43:1–43:12. SC '16, IEEE Press, Piscataway, NJ, USA (2016), `http://dl.acm.org/citation.cfm?id=3014904.3014962`, DOI: 10.1145/2063384.2063415

15. Wang, B., Ethier, S., Tang, W., Williams, T., Ibrahim, K.Z., Madduri, K., Williams, S., Oliker, L.: Kinetic turbulence simulations at extreme scale on leadership-class systems. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 82:1–82:12. SC '13, ACM, New York, NY, USA (2013), DOI: 10.1103/physreve.52.5646

16. Wang, Y., Lin, J., Cai, L., Tang, W., Ethier, S., Wang, B., See, S., Matsuoka, S.: Porting and optimizing gtc-p on taihulight supercomputer with sunway openacc. In: HPC China (2016)

17. Xiao, Y., Lin, Z.: Turbulent transport of trapped-electron modes in collisionless plasmas. Phys. Rev. Lett. 103, 085004 (Aug 2009), DOI: 10.1109/n-ssc.2006.4785860

18. Zhang, W.L.: CAAR project mid-term report. To be submitted (2016)