Runtimes (application facing)

QUESTIONS	XPRESS	TG X-Stack	DEGAS	D-TEC	DynAX	X-TUNE	GVR	CORVETTE	SLEEC	PIPER
PI	Ron Brightwell	Shekhar Borkar	Katherine Yelick	Daniel Quinlan	Guang Gao	Mary Hall	Andrew Chien	Koushik Sen	Milind Kulkarni	Martin Schulz
What policies and/or mechanisms will your runtime use to schedule code and place data for 100M objects (executing code, data elements, etc.) in a scalable fashion?	A hierarchical representation of logical contexts and tasks (processes and compute complexes) provides semantic representations of relative locality for placement of data objects and the tasks that are performed on them. Where data is widely distributed, they can be organized on separate processes distributed across multiple nodes with methods that allow actual work to be performed near the data. Research is exploring the allocation of resources by the LXK OS to the HPX runtime system and the policies to be implemented including programming interface semantics.	Open Community Runtime (OCR) will optimize for data-movement scalability. Our programming model divides an application into event-driven tasks with explicit data-dependences. Our runtime uses of this to schedule code close to its data or move the data close to the code. Scalability will be achieved through hierarchical task-stealing favoring locality.	The DEGAS runtime uses one-sided communication (put, get, active messages, atomics, and remote enqueue of tasks) to place data and work across a large-scale machine. Within a node there are currently two scheduling approaches being pursued. One (under HCLib/Habanero-C) is built on OCR and uses a dynamic task scheduler; it is being evaluated to determine the need for locality control within the node; the second is derived from the UPC runtime and has both a fixed set of locality-aware threads tied to cores (or hardware threads or NUMA domains – it's an abstraction that can be used a various machine levels), augmented with voluntary task scheduling for both locality and remotely generated dynamic tasks. A global task stealing scheduler is also part of the DEGAS plan and exists in prototype form; as with dynamic tasking, it is to be used on-demand for applications that are not naturally load balanced (e.g., divide-and-conquer problems with irregular trees).	(D-TEC) The APGAS (Asynchronous Partitioned Global Address Space) runtime uses a work-stealing scheduler to dynamically schedule tasks within a node. We are introducing areas to enable finer-grained locality and scheduling control within a node (Place). By design the runtime does not directly address automatic cross-node data placement. The APGAS runtime/programming model does provide primitive mechanisms (Places and Areas; at/async/finish) that allow application frameworks to productively implement data placement and cross-node scheduling frameworks on top of the runtime.	The SWift Adaptive Runtime Machine (SWARM) has a "locale" hierarchy, which roughly mirrors the hardware architecture hierarchy. Each locale has a set of local scheduler queues, allowing distributed and scalable scheduling. Data allocation and task/data migration are expressed to ensure proper parallelism around the conjunction. SWARM will rely on a single-assignment policy to prevent the need for globally coordinated checkout or write-back operations.	The compiler for X-TUNE must generate code with hierarchical threading, and will rely on the run-time to manage that threading efficiently. Point-to-point synchronization between threads may be more efficient than barriers to allow more dynamic behavior of the threads.	(GVR) will use performance information for varied memory and storage types (DRAM, NVRAM, SSD, Disk), resource failure rate and prediction, redundancy in data encoding, existing version data copies and their location, as well as communication costs to place data. GVR does not include code scheduling mechanisms.	(CORVETTE)	SLEEC does not have a true runtime component, except insofar as we are developing single-node runtimes to, e.g., manage data movement between cores and accelerators. We also perform small-scale inspector/executor-style scheduling for applications. However, we expect to rely on other systems for our large-scale runtime needs.	N/A
What features will allow your runtime to dynamically adapt the schedule and placement for 100K sockets to improve the metrics of code-data affinity, power consumption, migration cost and resiliency?	The HPX/LXK System software architecture (also known as the "OpenX Architecture" integrates a closed-loop introspection component comprising the APEX and RCR components within the runtime and OS respectively. Code-data affinity is supported by multiple mechanisms. Intra-compute complex (thread) function keeps all private or local data in the same locality. Parcels move work to the data when preferred although supports data access and gathers as appropriate. Processes keep shared data organized within a single logical context that can be spread across multiple localities. The effective reduction of latency effects also reduces data movement energy. For resiliency reconfiguration and recovery data migration is enabled by logical active global address space. Research is being performed to address these issues; some under other funding.	If the hardware supports it, OCR will monitor performance and power counters to adapt its scheduling and data-placement to better utilize the machine.	(DEGAS) For resilience, the DEGAS runtime uses customizable application and system-level policies to trade-off the storage costs associated with resilient processing against the expected failure rate, with an objective of optimizing the expected forward progress in an application against expected recovery and preservation times. The Containment Domain hierarchy allows the storage hierarchy of the system to be mapped to a hierarchical resilience structure. Process migration for GAS applications is planned, and we are investigating live migration techniques to move work around the system without stopping the application, or individual processes, from running during migration. The UPC language makes memory affinity explicit for programmers, and UPC supports teams as a construct to manage communications locality.	Automatic cross-socket migration and placement is not a topic we are actively exploring at the APGAS runtime level.	The locale hierarchy, runtime awareness of high-level data types, and support for task affinities to certain hardware will allow the runtime to make good placement decisions and move tasks and data around the system as needed to minimize the overall energy costs and improve efficiency. The use of a single-assignment data model and hints associated with particular tasks or data allows the runtime to establish good code-data affinity and energy efficiency.	Autotuning is the main mechanism that allows our project to adapt to execution context. In the long term, this autotuning must be performed during program execution to support dynamically-varying execution contexts.	(GVR) creates multiple-versions (snapshots) of globally-accessible data arrays as the primary basis or resilience. GVR creates an independent stream for each resilient data array, allowing it to be independently versioned, recovered, and managed - different from checkpointing -- and enabling a wealth of efficiency optimizations and flexible control by the application. Beneath that, GVR will optimize location, encoding, version creation and deletion, to maximize compute performance, resilience coverage, energy efficiency, and even wear-out lifetime of non-volatile storage devices (NVRAM).	(CORVETTE)	N/A	N/A
How will the runtime manage resources (compute, memory, power, bandwidth) for 100K sockets to meet a power, energy and performance objective?	The HPX runtime system maintains an abstraction of global data and compute complexes (threads) within the context of the ParalleX process hierarchy and engages in a bi-directional protocol with the LXK lightweight kernel to acquire and employ memory blocks and OS thread executables. As the OS manages multiple job program resource conflicts and the HPX runtime manages the intra-job task requirements and priorities, the two work together in dialog to balance the complex tradeoffs. Power imposes upper constraints at the node (locality) and socket level limited by the OS. Energy usage is governed by the ParalleX Side-Path Energy Suppression methodology that (attempts) to determine critical path of execution to which highest power is applied and reduces energy usage to the non-critical (side-path) work to the degree that the critical path does not change thus minimizing total energy with shortest time to completion. This strategy addresses scaling of both energy and performance objectives.	OCR will manage resources based on the application's needs and the power budget and turn off or scale back unneeded resources.	The DEGAS energy goals are primarily met by avoiding data movement both within and between nodes. "Communication avoidance" is a primary goal of the project in language, compilers and runtimes and has proven ties to energy use and performance. Dynamic energy management will be handled by the dynamic tasking on node and global task stealing between nodes, which as noted above is a voluntary and therefore "tunable" part of the runtime.	Using techniques developed in the SEEC runtime, the runtime could adaptively monitor application progress and increase/decrease resource utilization to minimize power consumption under the constraints of meeting application performance targets. This requires the application to be modified to report an abstract notion of progress to the runtime, the system software and hardware to provide the necessary monitoring APIs, and for the system software and hardware to provide the ability to dynamically adjust power consumption at the cost of reduced performance/reliability.	The runtime software will allocate only as many processors and as much memory as an application needs for efficient execution. It should be possible to adjust these parameters according to the real time or energy efficiency requirements indicated by the system user. Past power consumption can either be read out from supporting hardware or modeled based on software characteristics. This data, in conjunction with a system- or user-designated power budgets and hints associated with particular tasks and objects, will help the runtime decide when to focus work and data in a smaller area, allowing it to clock- or power-gate the remainder of the hardware, or when to spread work out across more of the system, requiring a higher power usage to induce a higher throughput. If hardware supports frequency scaling, this can be used to more finely tune power usage in runtime-managed components.	Autotuning can be used to support multiple objectives, as long as the tradeoff space and the goals of the developers are well understood. The question is really how much performance may be sacrificed to meet other optimization objectives.	(GVR) the optimization for resilience embodied in GVR - and its application partnership - can be constrained by power, energy and performance limits. The philosophy of GVR as a library is to adapt to these as external constraints, and is therefore compatible with a variety of runtime and programming system tools.	(CORVETTE)	N/A	N/A
How does the runtime software itself scale to 100K sockets? Specifically, how does it distribute, monitor and balance itself and how is it resilient to failures?	Individual instances of runtime system functions and responsibilities are created on a per node basis and per user program basis to spread the work uniformly as a system scales in workload (number of user jobs) and scales to larger number of hardware localities (ensembles of sockets). Introspection at the hardware support layer and software application layer detects and manages load balance through the RIOS control interface, the APEX runtime instrumentation and control layer, and the RCR black-boarding at the OS layer. Resiliency will be supported through the ParalleX execution model micro-checkpointing cross-cutting Compute-Validate-Commit cycle that employs hierarchical fault zones. This dynamic methodology engages all component layers of the hardware-software system for fault detection, isolation, diagnosis, reconfiguration, recovery, and restart.	OCR functionality is hierarchically distributed along the hardware's natural computation hierarchy (if it has one) or imposing an arbitrary one. OCR divides cores into "runtime" and "user". For efficiency, "user" cores run a small layer of the runtime and manage that specific core. The other "runtime" cores manage the user cores in a hierarchical fashion where the "runtime" cores "closest" to the "user" cores will perform low-latency simple scheduling decisions whereas higher level cores will perform longer-term optimization operations.	(DEGAS) DEGAS is already highly scalable on the largest machines available today and while some scaling issues in hierarchical synchronization (phasers), collective communication, and job startup require constant attention within the runtime, we do not see any major barriers to arbitrary scale. Note that the runtime is parallel by default (a job starts with a task per core/numa-domain/hardware thread) which greatly aids in scalability. Balancing due to resilience or load problems is done with the dynamic tasking and work stealing across nodes, both envisioned as voluntary within UPC++ and "by default" within a node in Habanero. We see this question of the default policy as key for the remainder of the project, but the same runtime mechanisms are needed in any case. Resilience is also in some sense tunable by the application using the general model of containment domains. GASNet-EX is designed to allow processes to fail and later be replaced. Distributing work is largely left to the applications programmer, but self-monitoring features and error reporting are being added to the interface to allow client runtimes to handle changes. We are investigating the semantic changes required to the GASNet-EX	The APGAS runtime has already been demonstrated to run non-resiliently and achieve scalable performance on a 55k core system. Most runtime operations are localized to a single APGAS place and thus naturally scale as the number of nodes increase. We have prototyped a resilient version of the APGAS runtime at a small scale (<500 cores) and are actively working on scaling the resilient version of the runtime to larger scale systems.	The runtime software will operate in all processor cores of the system, and will divide the system into executive and worker cores, with a hierarchy of executive cores associated with each non-leaf locale, managing each other and the workers. This helps localize work and data, but allows load to spill out into wider regions if a narrower region is flooded at any point. If the locale hierarchy is aligned with the hardware memory and communications hierarchy, it also helps localize the effects of any hardware failure. Detection of a failure that impacts correctness of runtime operation will result in the processor core, memory unit, or subsystem being taken out of service and the same operation being retried elsewhere. We will use containment domains to establish strict task boundaries, and can use earlier versions of data for resumption of failed tasks. Because we use a single-assignment model, the chance of overwriting old data from which recovery would otherwise be possible is eliminated and the extent of the effects of any failure can be limited.	This is not applicable to X-TUNE as we rely on a run-time system provided by others.	(GVR) is based on a decentralized architecture that replicates metadata across the machine, and creates redundant data versions for application resiliency. The GVR architecture will exploit replicated metadata storage, and a stateless recovery architecture to enable resilience to scale from application thru GVR implementation resiliences as well as from petascale to exascale systems.	(CORVETTE)	SLEEC's runtimes are intended to operate within the scope of a single node, or at a small scale. We rely on other runtimes to provide higher levels of the hierarchy.	The PIPER concepts includes performance and correctness data collection across the entire system software stack - this does require runtime support to cleanly and scalably aggregate and process data. This can be combined with work in the Visualization/Data Area.
What is the efficiency of the runtime? Specifically, how much impact does the runtime have on a) the total execution time of the application and b) resources taken from algorithmic computations? What are your plans to maximize efficiency? How will runtime overhead scale to 100K sockets?	The HPX runtime is event driven and stays out of the way of the user codes executing intra-thread for purposes of efficiency. However, inter-thread there are a number of overhead actions that impact efficiency and impose a lower bound on thread granularity, which limits scalability for fixed size workloads. OS overhead (LXK) is fixed on a per node basis and therefore scalable. HPX process calls across nodes (conceptually millions) employ symmetric semantics (synchronous versus asynchronous) for portability, parcels for message-driven computing in combination with local control objects to manage asynchrony including mitigation of latency effects, and active global address space to handle remote data load and stores. Research will determine the scaling factors for these as well as the time and energy efficiencies that may be achieved.	OCR code runs on cores that are physically separate from those for user code. Our goal is to have enough "runtime" cores that runtime overhead is completely masked by the application code. As machine size increases, more runtime cores will be needed to handle higher-level functions and global optimizations but this will increase very slowly.	(DEGAS) As noted above, we see no major barriers to scaling to arbitrary machine sizes, but expect resource management at this scale to require additional research and engineering. The large number of cores on a node accessing a shared communication resource is one such problem. The dynamic tasking runtimes have more overhead, but we are working to minimize the difference in performance between the static and dynamic case. In our experience the major problem is lose of locality from the dynamic case, which we are addressing in various ways, including an "inspector-executor" style scheduler.	Running at 55k cores on typical kernel benchmarks, the APGAS runtime has been demonstrated to have very low overheads. As a general design principle, the runtime overheads should be expected to be proportional to the frequency with which the application requests services from the runtime.	We have focused very heavily on minimizing the amount of inline work, allowing the application to run un-hindered, and minimizing the hardware resources required for the runtime-internal threads. Overall, we anticipate that the processing overhead per core is expected to be essentially constant regardless of system size, with the exception of global operations like barriers and reductions which may require additional time scaling with the logarithm of the system size. Memory usage for thread descriptors and stacks will be linear with the number of cores, although temporary linearitmic (i.e., O(n ln n) for n coordinating agents) memory blocks may be needed to manage global operations. Very little static-/runtime-bound data is required beyond that, aside from what's required for basic interfacing with the underlying platform.	This is not applicable to X-TUNE as we rely on a run-time system provided by others.	(GVR) seeks to minimize resilience overhead. We have performed experiments with numerous applications (ddcMD, OpenMC, PCG, GMRES, and mini-apps such miniFE, miniMD) that demonstrate overheads of less than 1% runtime without any special hardware support. With novel emerging features such as storage-class memory (integrated NVRAM), we expect this overhead to be even smaller.	(CORVETTE)	Because SLEEC focuses on small-scale runtimes that are directly integrated with application code, we expect our runtime overheads to be negligible and, essentially, independent of scale (because scaling will be provided by other runtime systems.	N/A
Do you support isolation of the runtime code from the user code to avoid violations and contamination?	The ParalleX process construct and hierarchy with capabilities addressing separates runtime functions from user functions. The global addressing permits runtime system instances to manipulate user "compute complexes" (e.g., threads) as first class objects. Independent runtime instances isolates multiple user applications sharing any particular localities (nodes). Research is exploring the costs and completeness of these protection mechanisms.	The majority of the runtime code runs on cores that are physically separate from the ones on which user code is running. Although we are currently considering a model where all cores can touch data everywhere else, our model will support possible hardware restriction (user cores cannot touch data in runtime cores).	(DEGAS) The runtime code is separate from user code, but there is no enforced isolation.	We are not directly addressing this issue. The design point we are pursuing is that there will be different instances of the APGAS runtime for different programs and isolation will be provided by other layers of the stack.	Complete isolation depends heavily on hardware support. For architectures which support it, such as Tralelka Glacier, SWARM isolates resources rather well, with only a thin shim layer of SWARM residing on the application cores. When possible, runtime decisions happen on executive cores, which have visibility and control over worker cores, but not vice-versa. When using hardware that does not support this kind of work division, it may not be possible to prevent violation/contamination over application cores. (When features like segmentation or virtual memory are present, these can potentially be used to enforce separation between the runtime and application code, but doing so may impose an enormous overhead---comparable to placing the runtime in the OS kernel---and this will likely not be worth the enormous performance hit that would be taken.)	We rely on run-time systems provided by others, and would simply invoke the run-time.	(GVR) supports use of operating system or other runtime mechanisms for this isolation, but provides no mechanisms of its own. GVR's design and implementation supports flexible recovery from detected violations or contamination.	(CORVETTE)	SLEEC's runtimes are application/domain-specific and hence intended to closely couple with the application code.	(PIPER)
What specific hardware features do you require for proper or efficient runtime operation (atomics, DMA, F/E bits, etc.)?	There are no absolute requirements for proper operation of the HPX runtime system beyond those found on conventional parallel and distributed systems. These include compound atomic operations, message exchange between nodes, scheduling of threads and their precise interrupts, and local virtual address translation. However, there are additional features that may be incorporated in the future that would dramatically reduce overheads, mitigate latencies, increase parallelism, and circumvent hotspots. Among such mechanisms for efficient runtime operation are hardware support for 1) user lightweight thread creation, termination, and context switching (including preemption), 2) global virtual address translation, 3) 'struct' processing for simultaneous multi-word processing (for local control objects among others), message driven computation, and combined DMA plus synchronization. Research will ascertain, evaluate, and analyze to degree of operational improvements that may be derived from such hardware support.	OCR requires hardware to support some form of atomic locking. Additional HW features identified for increased efficiency: 1) Remote atomics for cheaper manipulation of far-away memory; 2) Heterogeneity to taylor "user" cores for user code and "runtime" cores for runtime code (no FP for example) 3) Fast runtime core-to-core communication to allow the runtime to communicate efficiently without impacting user code 4) Asynchronous data movement (DMA engines) 5) HW monitoring to allow introspection and adaptation; 6) knowledge of HW structure (memory costs, network links available, etc) enabling more efficient scheduling and placement.	(DEGAS) For efficient processing, remote DMA operations and low computational overhead queue pair access are essential, as is interrupt-free message processing. Loose ordering restrictions on message and RDMA processing are important, as is the ability to issue relatively large numbers (hundreds) of outstanding remote memory operations. Registration and pinning of RDMA memory often remain issues for us, and we would like both low-overhead memory registration techniques and the ability to have large numbers of registered memory areas, as we have encountered difficulties due to limitations on the number of (distinct) memory regions that can be accessed by an RDMA-capable device.	Nothing beyond those found already found on conventional parallel and distributed systems. The APGAS runtime has been extended to exploit unique hardware capabilities of particular machines (eg Torrent exploitation in Power775) and if unique hardware capabilities are available on future systems the APGAS runtime could be extended to exploit them.	Our fundamental requirements are atomic operations (preferably at least compare-and-swap), memory fences, RDMA, power/clock/frequency management, and hardware failure event notification. Optionally, F/E bits, and explicit associative memories would result in additional efficiency improvements. If a transparent data caches is available on each core, then features like hardware transactional memory can greatly speed up operations on shared data structures.	Autotuning relies on accurate hardware monitoring to provide measurements used to calculate optimization objectives.	(GVR) is designed for portability, and should be able to run on systems ranging from current-day petascale to CORAL to Exascale systems. However, hardware features such as integrated NVRAM, efficient change tracking, data compression, efficient and reliable RMA/RDMA, collectives, etc. will further increase the efficiency of GVR.	(CORVETTE)	N/A	Yes, that should be the case.
What is your model for execution of multiple different programs (ie: a single mention would be doing more than one thing) in terms of division, isolation, containment and protection?	The HPX runtime system supports the ParalleX processes, which serve as logical contexts and are referenced through a hierarchical namespace. The global root process of the entire system provides global naming. Each program has a program root process that contains the instances of the dedicated runtime kernel and the "main" process of the user applications. The process boundaries incorporate a form of capabilities based addressing for protections. Programs are logically separate and isolated although can interact through the upper hierarchy of the process stack. Nonetheless, programs may share physical resources (localities). The underlying OS manages the protections of the virtual address space.	Our programming model splits user code into small event-driven tasks (EDTs). Multiple non-related EDT sub-graphs can coexist at the same time with the same runtime. While not isolating applications, it does automatically globally balance all the applications at once. The locality aware scheduling will also naturally migrate related data and code closer together thereby physically partitioning the different applications. If a more secure model is required, different runtimes can run on a subset of the machine thereby statically partitioning the machine for the various applications; it is more secure but less flexible.	(DEGAS) Our model is support for hierarchical applications, at which the top level hierarchy may be logically separate programs (or physics models, or…). We also have a strong emphasis on interoperability, including with current MPI, MPI+X applications. Part of our interoperability between tasking layers is supported by the work on Lithe. We are interested in the iPython model for combining applications into workflows, but this is not part of the DEGAS project itself.	If this is another way of asking question 6), then this is not an issue being addressed by our runtime work.	The SWARM runtime depends on isolation features in the hardware, which vary from one hardware platform to the next. Where possible, SWARM will make use of hardware isolation features to protect multiple programs from each other. If a single instance of the SWARM runtime software is in control of all of the running programs which must be isolated from each other, it is both well-suited and in a perfect position to ensure that no program starves others for resources. If there is a higher level OS or executive managing multiple SWARM instances, there may need to be higher-level signaling to prevent resource starvation. When distinct applications must be run within the same runtime, hardware features will be required to prevent the applications from reading from or writing into each other's state; SWARM cannot provide this guarantee on its own. However, instituting a single-assignment policy helps prevent most applications from butting up against the hardware protection mechanisms accidentally---doing so constitutes programmer error with regard to the application. If distinct applications have distinct runtime instances, then SWARM has little to no control its client applications' attempts to read/write things they shouldn't, and so must rely entirely on hardware mechanisms and lower software layers for protection.	N/A	(GVR) Each program would have a unique instance of the GVR library, and as such create a version store that includes several independent streams of global-view structures (tuned for efficiency and resilience). However, the version stores do not interact with each other, so there is no interference. Access control to version stores is enforced by the operating system.	(CORVETTE)	N/A	N/A