XPRESS: eXascale PRogramming Environment and System Software

XPRESS

eXascale PRogramming Environment and System Software

Final Report: September 1, 2012 – August 31, 2017

Submitted: July 14, 2017

The XPRESS project is a collaborative effort across eight institutions that include laboratories and universities. Overall leadership for the integrated project is provided by Sandia National Laboratories. Other laboratories involved are Oak Ridge National Laboratory and Lawrence Berkeley National Laboratory. University partners include Indiana University, Louisiana State University, the University of Oregon, the University of North Carolina at Chapel Hill, and the University of Houston.

Office of Science
NNSA National Nuclear Security Administration

XPRESS: eXascale PRogramming Environment and System Software

Overview

This Final Technical Report constitutes the completion of Deliverable “xxx“ for the XPRESS supported by the Department of Energy under Award Number(s) DE-SC0008809 initiated in September 2012.

Introduction

The innovative system software stack of XPRESS is poised to enable practical and useful exascale computing by directly addressing the critical computing challenges of efficiency, scalability, and programmability through introspective dynamic adaptive resource management and task scheduling.

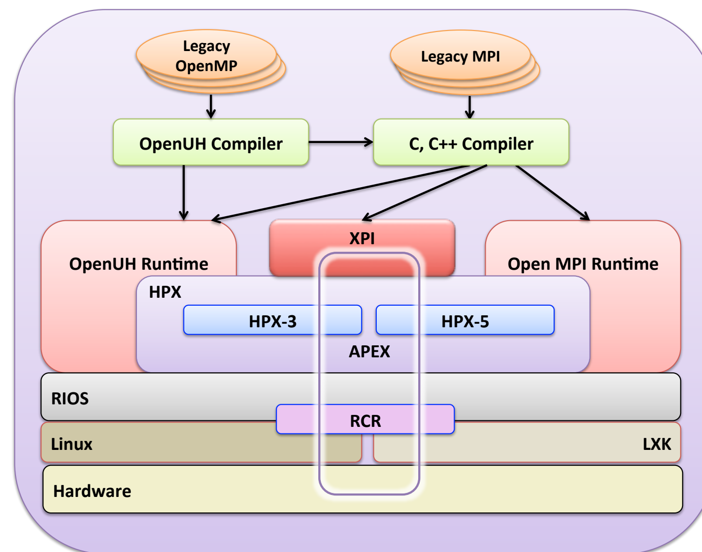


Figure1: XPRESS System Architecture

Features. Specific components of the XPRESS System Architecture include:

- ParalleX: Indiana University - Cross-cutting execution model of system co-design
- LXK – Lightweight eXtreme-scale Kernel (Kitten): Sandia National Laboratories - Fourth-generation scalable compute node operating system
- HPX runtime system software: Louisiana State University, Indiana University - Supports introspection for guided computing through dynamic adaptivity
- Autonomic Performance Environment for Exascale (APEX), Resource Centric Reflection (RCR) – application introspection: University of Oregon, RENCI - A derivative of TAU instrumentation and monitoring software system, integration of low-level system data acquisition
- DOE applications: LBL, ORNL, SNL - Drives co-design and performance studies relevant to DOE
- RIOS: Interface between the operating system and the runtime system
- Conventional Programming Interfaces for legacy codes and skill sets: University of Houston, Stony Brook - MPI, OpenMP

BERKELEY LAB LSU OAK RIDGE National Laboratory Sandia National Laboratories

- XPI: User programming syntax and source-to-source compiler target for high-level programming languages

Activities and Findings

ParalleX Execution Model

ParalleX is a cross-cutting many-tasking execution model that supports message driven computation and uses dynamics adaptive methods to manage asynchrony and enable scalable operation on large systems. It is an evolving parallel execution model derived to exploit the opportunities and address the challenges of emerging technology trends. It aims to ensure effective performance gain into the Exascale era of the next decade by addressing the challenges of starvation, latency, overhead, waiting, energy and reliability (SLOWER). Starvation reflects an insufficiency of concurrent work to keep all the critical sources engaged. Latency quantifies the response time distance for remove access and service requests. Overhead distinguishes the work needed to manage parallel resources and task scheduling which does not contribute to the useful computational work itself. Delay is due to waiting time incurred due to resource access conflicts. Energy and its rate of consumption is a key parameter in the raw sequential performance potentially affecting both logic voltage and clock rate. Reliability impacts availability of the system and therefore the percentage of time the system can be employed for useful work.

The primary semantic components of ParalleX includes:

- *Locality* – an encapsulation of resources in a synchronous domain that guarantees bounded access and service request time as well as compound atomic sequence of operations
- *Global Name Space* – provides the semantic means of accessing any first class objects in a physically distributed applications
- *ParalleX Processes* – an abstraction of distributed context hierarchy integrating data objects, actions, task data, and mapping data
- *Compute Complexes* – a generalization of thread complex
- *Local Control Objects (LCOs)* – provides synchronization, management of parallelism, and migration of continuations
- *Parcels* – message driven computing that combines data transfer and event based information using a variant of active messages to permit the management of distributed flow control in a context of asynchrony

At the core of the ParalleX strategy is a new framework to replace static methods with dynamic adaptive techniques. This method exploits runtime information and employs unused resources. It benefits from locality while managing distributed asynchrony. ParalleX is a crosscutting model to facilitate co-design and interoperability among system component layers from hardware architecture to programming interfaces. ParalleX addresses starvation by increasing the amount of parallelism available and facilitating work distribution, by incorporating coarse, medium and fine grain parallelism semantics. ParalleX processes (Coarse-grained parallelism), can span multiple localities, share nodes with other processes

and migrate if necessary. ParalleX organizes the actual work it performs as medium grain compute complexes, which often are performed as threads and permit context switching for non-blocking of physical resources. LCOs provide declarative constraint based synchronization and scheduling to increase parallelism by control relationship and dynamic adaptivity. With LCOs, ParalleX largely eliminates over constraining global barriers, enabling overlap of successive phases of an evolving computation and exposing additional form of parallelism. Fine grain parallelism is given in terms of static data flow control semantics within the context of a compute complex.

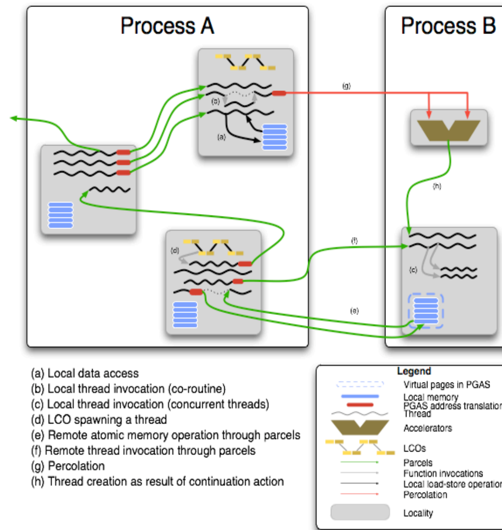


Figure 2: This diagram illustrates many of the key semantic constructs and mechanisms of the ParalleX model of computation

ParalleX addresses latency through both mechanisms to reduce its magnitude and hiding its effects. The context switching permits a thread that is waiting for a long latency request to be replaced by a pending task ready to do work, overlapping computation with communication. Multithreading of work trades parallelism for latency hiding. Parcels move work to the data, not always requiring data to be gathered to a fixed location of control. Latency of the data accesses is reduced through the reduction of number and size of global messages. The use of LCOs provides event-driven control simultaneously addressing latency of action at a distance and managing the uncertainty of asynchrony of long latency operations.

ParalleX reduces overhead through LCOs that support event driven computation control to minimize wasted work and avoid overly constraining semantics. The global barrier is eliminated through ParalleX. ParalleX uses lightweight user threads in lieu of heavy weight OS threads (e.g., pthreads) that reduce the overhead of context switching time.

ParalleX supports dynamic adaptive means by which, the runtime can allocate a different available resource with similar capabilities (e.g., rerouting of network traffic where multiple paths exists, distribution of synchronization elements, multiple physical threads for task

execution). The event driven distributed flow control eliminates polling and reduces the number of sources of synchronization delays.

Experimental Results

Simulations were conducted to look into the effects of overhead on the performance of an abstract machine running a program with a particular instruction mix. The results of the simulation are illustrated in Figure 3. The graph presented shows the performance impact when lightweight user level threads are multiplexed on OS threads to provide a significant performance advantage in hiding network latency.

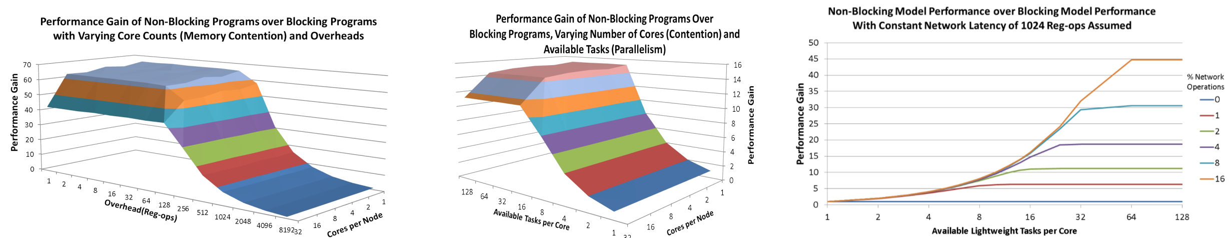


Figure 3: Performance impact of lightweight user level threads multiplexed on OS threads

Formal Semantics for ParalleX

Exascale execution models are large complex specifications defining the delicate interrelations of their component layers and governing their interoperability. Defining execution models is error prone, which may leave undetected inconsistencies and may be incomplete. Furthermore, execution models are hard to communicate effectively to programmers, and researchers. This problem is addressed through the means of formal analysis. Formal semantics is a rigorous, mathematically defined, precise description of the execution model. A formal semantics helps to detect the design mistakes early and ensure the completeness of the specified behavior. It helps to communicate the model effectively and opens the possibility of proving programs correct.

Recent Results

We have developed an operational semantics for a core fragment of ParalleX. Such a core fragment covers the main interesting computational means of ParalleX and is large enough for allowing writing meaningful computation. The overall semantics maps programs to *automata* whose states are run-time configurations of a ParalleX system. These *automata* are defined by means of *conditional rules*, such as the one shown in the figure below just as an example. Without going into details, conditional rules specify a precise relation among the mathematical abstractions that we use for memories, buffers, network and the various components that ParalleX employs for carrying out synchronous and asynchronous computation.

Not only such a semantics can be the subject of proofs, it also can be loaded into specialized tools for advanced formal analysis, such as theorem provers, model checkers and specification systems that support automatic tests. We have a preliminary implementation of the formal semantics, which is executable and can also be injected into a theorem prover. Carrying out

advanced formal analysis on the semantics of ParalleX is part of our future work. Furthermore, we have devised a type system for ParalleX that rules out data races.

$$\textbf{if}\quad \begin{array}{l} M = S(\text{memory}), M(a) = c \\ z \leftarrow x + y \in c(\text{control}) \\ c(\text{locals})(x) = n_x \\ c(\text{locals})(y) = n_y \end{array}$$

$$\textbf{then}\quad S \longmapsto S'$$

$$\textbf{where}\quad \begin{array}{l} S' = S\{\text{memory} \mapsto M\{a \mapsto c'\}\} \\ c' = \{\text{control} \mapsto c(\text{control}) - \{z \leftarrow x + y\}, \text{locals} \mapsto L'\} \\ L' = c(\text{locals})\{z \mapsto n_x + n_y\} \end{array}$$

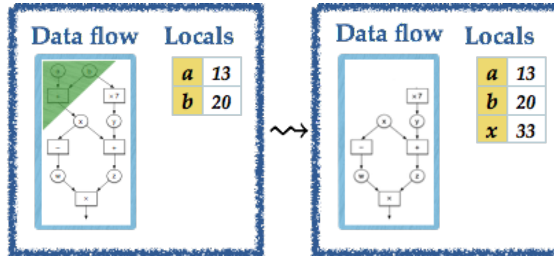


Figure 4: Excerpt from the operational semantics

The shift to eXascale computing promises a high impact on science and industry. Formal Semantics will place eXascale computing on a firm foundation, enabling a more rapid and confident shift to eXascale built upon a reliable and robust infrastructure.

- Operational semantics for a core fragment of ParalleX
- Proposal type system for ruling out data-races
- Executable prototype implementation of formal semantics

HPX-5 Runtime System

HPX-5 [1] is a state-of-the-art runtime system for extreme-scale computing. Version 4.1.0 of the HPX-5 runtime system, which was released on May 9, 2017, represents a significant maturation of the sequence of HPX-5 releases to date for efficient scalable general purpose high performance computing. It incorporates new optimization for performance, features associated with the ParalleX execution model, and programmer services including C++ bindings and collectives.

HPX-5 is a realization of the ParalleX execution model, which establishes the runtime's roles and responsibilities with respect to other interoperating system layers, and explicitly includes a performance model that provides an analytic framework for performance and optimization. As an Asynchronous Multi-Tasking (AMT) software system, HPX-5 is event-driven, enabling the migration of continuations and the movement of work to data, when appropriate, based on sophisticated local control synchronization objects (e.g., futures, dataflow) and active messages. ParalleX compute complexes, embodied as lightweight, first-class threads, can block, perform global mutable side effects, employ non-strict firing rules, and serve as continuations. HPX-5 employs an active global address space in which virtually addressed objects can migrate across the physical system without changing address. First-class named processes can span and share nodes.

HPX-5 is an evolving runtime system used both to enable dynamic adaptive parallel applications and to conduct path-finding experimentation to quantify effects of latency, overhead, contention, and parallelism of its integral mechanisms. These performance parameters determine a trade-off space within which dynamic control is performed for best performance. It is an area of active research driven by complex applications and advances in HPC architecture. HPX-5 employs dynamic and adaptive resource management and task scheduling to achieve the significant

improvements in efficiency and scalability necessary to deploy many classes of parallel applications.

HPX-5 is ported to a diverse set of systems is reliable and programmable, scales across multi-core and multi-node systems, and delivers efficiency improvements for irregular, time-varying problems.

HPX-5 is used for a broad range of scientific applications, helping scientists and developers write code that shows better performance on irregular applications and at scale when compared to more conventional programming models such as MPI. For the application developer, it provides dynamic adaptive resource management and task scheduling to reach otherwise unachievable efficiencies in time and energy and scalability. HPX-5 supports such applications with implementation of features like Active Global Address Space (AGAS), ParalleX Processes, Complexes (ParalleX Threads and Thread Management), Parcel Transport and Parcel Management, Local Control Objects (LCOs) and Localities. Fine-grained computation is expressed using actions. Computation is logically grouped into processes to provide quiescence and termination detection. LCOs are synchronization objects that manage local and distributed control flow and have a global address. The heart of HPX-5 is a lightweight thread scheduler that directly schedules lightweight actions by multiplexing them on a set of heavyweight scheduler threads.

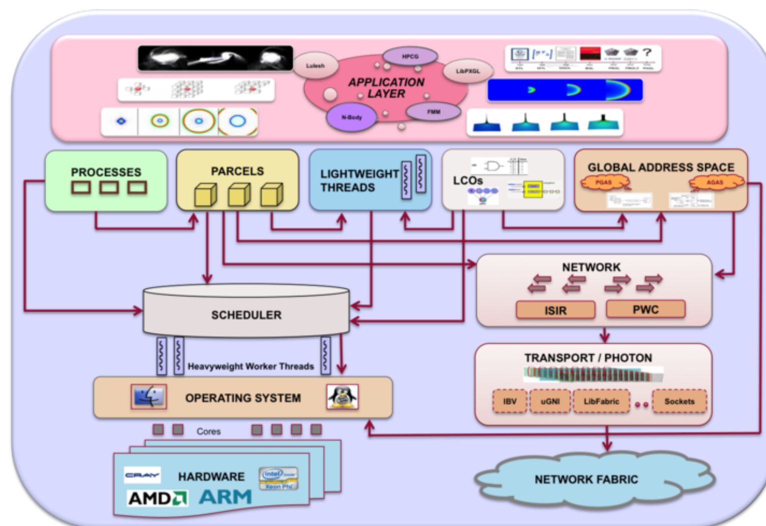


Figure 5: Synopsis of the HPX-5 runtime system.

Features of HPX-5

- Fine-grained execution through blockable **lightweight threads** and unified access to a global address space.
- High-performance PGAS implementation which supports low-level one-sided operations and two-sided active messages with continuations, and an experimental **AGAS** option with active load balancing that allows the binding of global to physical addresses to vary dynamically.

- Makes concurrency manageable with globally allocated **lightweight control objects** (LCOs) based synchronization (futures, gates, reductions, dataflow) allowing thread execution or parcel instantiation to wait for events without execution resource consumption.
- Higher level abstractions including asynchronous remote-procedure-call options, data parallel loop constructs, and system abstractions like timers.
- Implementation of **distributed processes** providing programmers with flexible termination detection and virtualized collectives.
- **Photon** networking library synthesizing RDMA-with-remote-completion directly on top of uGNI, IB verbs, or libfabric. For portability and legacy support, HPX-5 emulates RDMA-with-remote-completion using MPI point-to-point messaging.
- Programmer services including C++ bindings and collectives (prototype non-blocking network collectives for hierarchical process collective operation).
- Leverages distributed GPU and co-processors (Intel Xeon Phi) through experimental **OpenCL** support.
- **PAPI** support for profiling.
- Integration with **APEX** policy engine (Autonomic Performance Environment for eXascale) support for runtime adaption, RCR and LXK OS.
- Migration of legacy applications through easy interoperability and co-existence with traditional runtimes like MPI. HPX-5 4.1.0 is also released along with several applications: LULESH, Wavelet AMR, HPCG, CoMD and the ParalleX Graph Library.

Localities

A **locality** is a distinct virtual address space in which an instance of the HPX–5 runtime exists. On a clustered system, each physical node usually has a single locality, though this is determined at runtime by the application launcher—oversubscription caused by the launcher may result in one locality per NUMA domain, or one locality per subset of cores. It is quite possible to write HPX-5 programs without explicitly referencing localities, but it sometimes helps to know they exist (for example when using the "PGAS" model for global memory, or when initializing C/C++ per-process global data).

Global Memory

An HPX–5 applications has three distinct regions of memory.

- **Local** memory is the memory that is normally returned by an allocator like malloc(). Local virtual addresses can be shared between lightweight threads within a locality, e.g., for parallel for operations, however local virtual addresses cannot be passed across localities. Local addresses are also used in the global memget/memput API for asynchronous data transfer.
- **Global** memory is memory allocated via the HPX–5 runtime and that has an address that can be shared among the entire HPX–5 applications. All global virtual addresses have a local virtual alias somewhere in the system, however a global virtual address does not necessarily have a mapping at the current locality. Global memory can be accessed through an untyped, asynchronous, memput-memget API. Alternately, when a parcel targets a global address, its thread will preferentially run at the locality at which the address is mapped in which case an application can pin the memory and interact directly with the data through its local aliases.

- **Registered** memory, allocated through the hpx_malloc_registered interface, is local memory that is optimized for use as the local address for the memget/memput operations. Lightweight thread stacks, and pinned local aliases, are implicitly registered.

HPX–5 supports several different memory implementations. These are largly transparent to the user allowing most code to run unmodified on a number of different archictures which is key to programmability and performance portability in HPX-5.

- Under the **SMP** implementation, which is only available when HPX–5 is running on a single node, global memory is effectively the same as local memory, except that the memory has a global address the same as it would in one of the normal HPX–5 memory models.
- Under the **PGAS** (**Partitioned Global Address Space**) implementation, global memory is allocated and managed similarly to how UPC does so. Memory can be allocated at the current locality or can be allocated cyclically over a number of localities.
- Under the **AGAS** (**Active Global Address Space**) implementation, global memory is allocated similarly to how it is allocated under the PGAS model, but it is not necessarily located at a fixed locality; it may be moved to other localities to balance system memory load or workload.

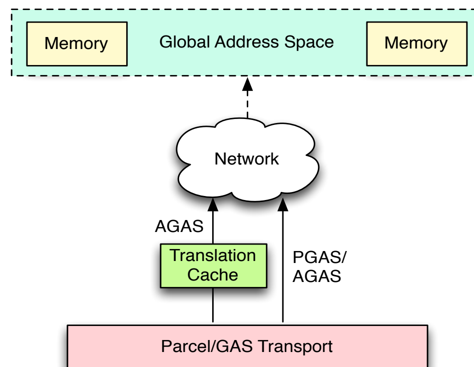


Figure 6: Coexisting AGAS and PGAS protocol stacks in HPX-5. AGAS implementation requires additional address translation component compared to PGAS; minimization of translation overhead is critical to obtaining high level of performance.

Within a locality applications may specify a soft core affinity for global addresses using the hpx_gas_set_affinity() interface. Threads generated from parcels targeting a global address with soft-core affinity will be preferentially scheduled on the requested core. Soft GAS affinity provides support for deep memory hierarchy while supporting high performance load balancing through workstealing.

- Soft affinity is of most use for well-balanced applications with long-running threads.
- Soft affinity is not a guaranteed affinity and programmers should make no assumptions about where threads will actually run.

The current libhpx implementation ships with three implementations of GAS affinity, which can be selected at runtime using the --hpx-gas-affinity option.

- The default --hpx-gas-affinity=none implementation ignores the affinity hint entirely. We intend this default to change in the future.
- The --hpx-gas-affinity=urcu implementation uses the userspace read-copy-update library and associated hash table to manage soft affinity bindings.
- The --hpx-gas-affinity=cuckoo implementation uses a libcuckoo hash table to manage soft affinity bindings.

The userspace read-copy-update implementation is the preferred option due to its low overhead for readers, but is currently not buildable for Darwin platforms, which can use libcuckoo instead.

Lightweight Threads & Actions

Actions are globally callable functions and are the thread entry points for HPX–5 lightweight threads (and tasks and interrupts). They are invoked through the use of an active-message parcel or through one of the higher-level remote procedure call interfaces. Actions are executed by HPX–5 lightweight thread and are usually bound to execute local to a specific **global addresses**; this is used by the HPX–5 runtime to schedule the action in the physical location that will be most efficient. HPX–5 threads are scheduled cooperatively and may block waiting for lightweight synchronization objects (LCOs).

LCOs

Synchronization and data-driven communication between threads is done via **LCOs** (**Lightweight Control Objects**). There are a variety of LCOs, and some have rather different characteristics but all LCOs may be waited on by threads and parcels until some condition is true or until the LCO has been **set** with a value (either by another action, or possibly by some other means). Many LCOs will have a value that can be retrieved by **get** actions.

The most commonly used kind of LCO is a **future**. A future can be waited on by one or more thread until it is set (it may be set with a value, or not). Many asynchronous functions in the HPX-5 API take a future as a parameter, and will signal completion via the future.

Another commonly used LCO is the ***and*** **LCO**. When an "and" LCO is created, it requires a parameter specifying how many times it may be set. Then multiple actions may set it one or more times, and a thread waiting on it will be released when it has been set as many times as was specified. and LCOs are often used in parallel loops.

Performance of HPX-5

Performance optimization of the HPX runtime system is one of the primary foci of the development effort. Introduction of new features as well as solutions intended to reduce the existing overheads are constantly gaged against the performance data collection integrated with Jenkins continuous integration tool. This is done to assess the viability of the implemented solutions, detection of compilation errors on multiple architectures, and regression testing.

MicroBenchmarks

The scalability of HPX–5 at relatively modest scales is highlighted through the speedup that was achieved for the applications that were ported to use HPX–5: DASHMM, LULESH, WAMR, HPCG and SSSP (ParalleX Graph Library, or libPXGL). However, to quantify the execution overheads involved in a distributed runtime system like HPX-5, a set of shared and distributed-memory micro-benchmarks, synthetic benchmarks and other performance tests have been run. Shared-memory performance should be comparable to leading shared-memory systems and distributed-memory performance should be comparable to leading distributed-memory implementations (e.g., MPI).

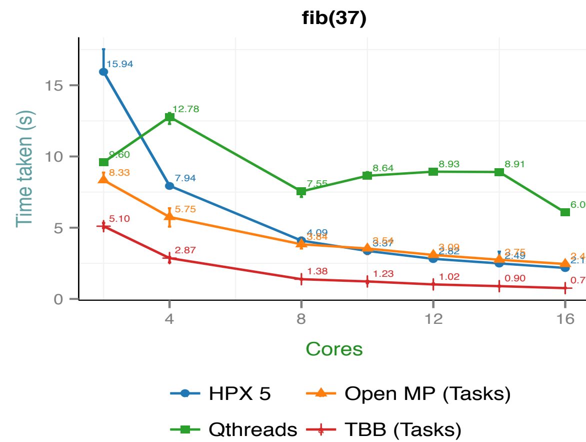
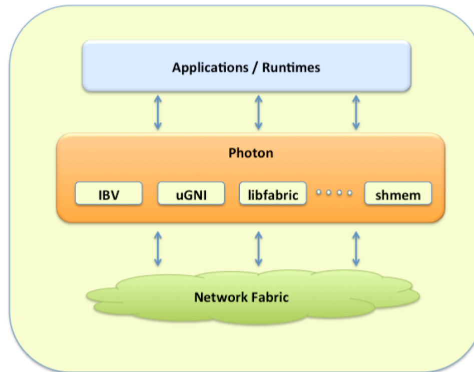


Figure 7: On-node Scheduler performance

To measure shared memory load balanced performance, we used a Fibonacci benchmark to assess the overhead of task creation and synchronization overheads in task-parallel runtime systems. Performance results comparing HPX to other systems are displayed in Figure 7. HPX-5 demonstrates similar speedup on multiple cores to the other systems. Performance measurements were performed on the IU Cutter cluster featuring 16-nodes, each equipped with 32GB memory and dual 8-core E5-2670 Intel processors (by far the most common 64-bit general-purpose processor currently deployed in supercomputing systems) clocked at 2.6GHz.

Photon RDMA Network Library

Photon provides consistent remote direct memory access (RDMA) semantics over multiple interconnect technologies such as InfiniBand and Cray's Aries and Gemini fabrics. Its goal is to minimize latency and maximize throughput for high-performance applications and runtime systems that can benefit from distributed, direct memory operations over a network. Memory management and asynchronous progress are exposed at a fine granularity, decoupling data transfers from the notification path. Importantly, a photon implements a pattern named "put-with-completion" (PWC) that optimizes for a general completion identifier mechanism in support of active message style computation.



- Figure 8: The diagram shows the high-level layering of HPX and Photon, with PWC being the primary, default network in HPX-5.

Features

PUT and GET with completion ID (CID)

- Variable length identifiers encode actions to execute on completed data operations
- "Packed" mode for small messages
- Switch to "2-PUT" mode at configurable threshold, using the Photon ledger

One-sided and rendezvous interface

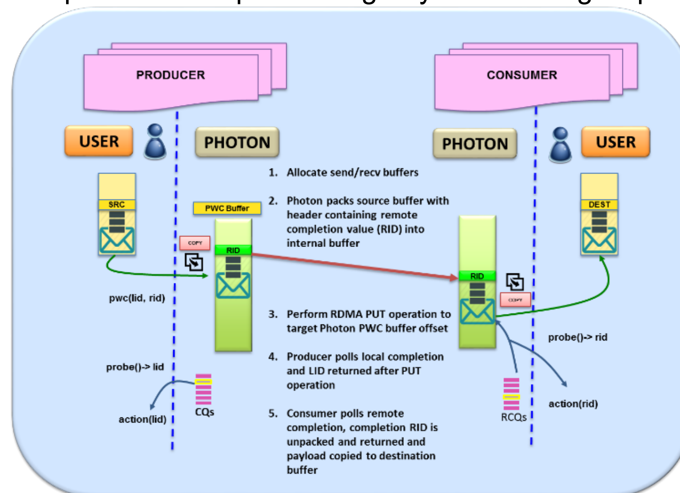
- Explicit control of completion ID probing suitable for AMTs such as HPX-5

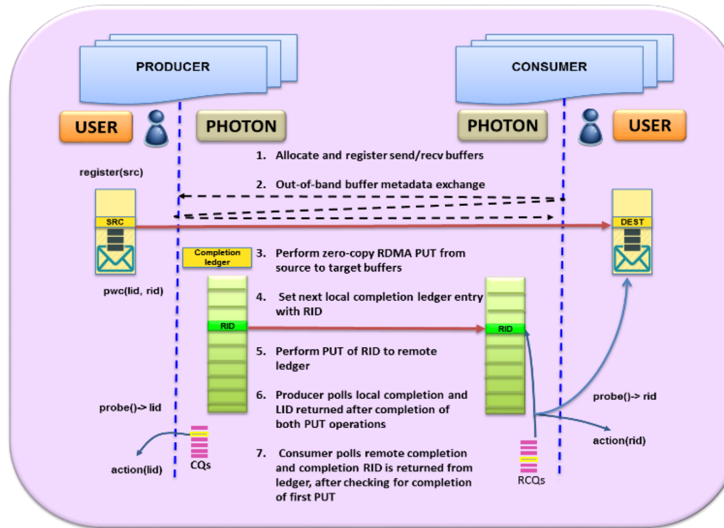
Thread safety for one-sided operations

- libsync concurrent data structures (HPX-5)
- Moving to native library atomics

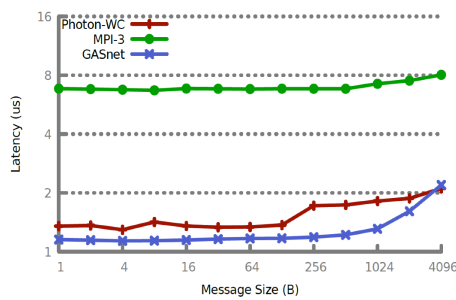
Request queuing with flow control

- Completion progress is queried while processing any outstanding requests

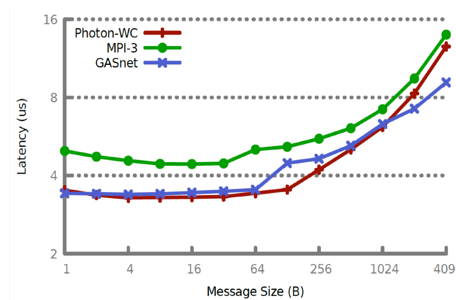




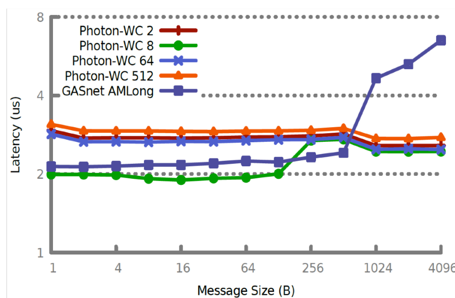
Cray Aries uGNI PUTs



Mellanox CX-3 RoCE PUTs



Variable CID Size



Scaling to 2048 nodes (49152 cores) on

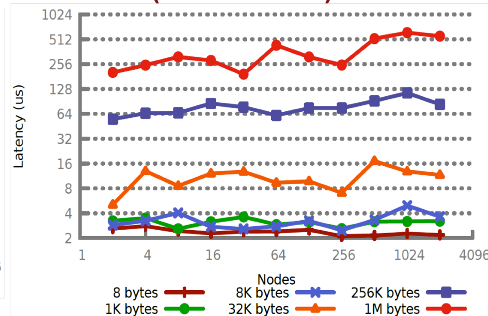
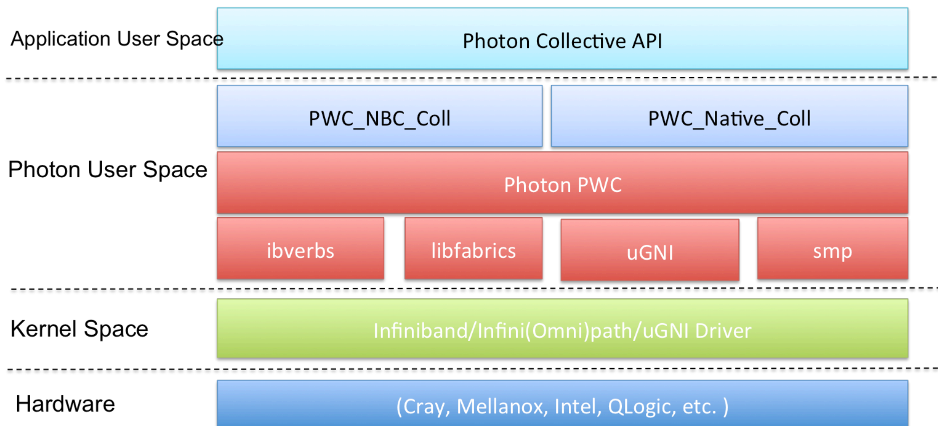


Figure 9: Performance of Photon - 8 byte CIDs (16 byte PWC overhead, 28 byte overhead for packed), 128 byte packed message threshold

Currently there are two implementations of Photon collectives

- libNBC based (NBC) – Non Blocking Collective library for asynchronous collectives
- Photon Native (PWC) – Experimental/highly optimized (Photon-specific optimizations) using completion path notifications



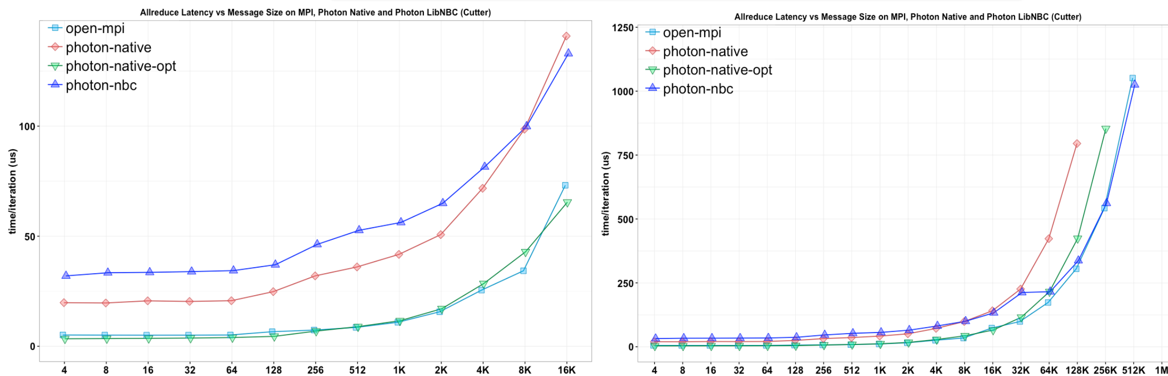


Figure 10: Experimental results on Cutter (Mellanox Connect X-3 HCA/16 nodes) with small message performance beginning to improve upon OMPI

Path Forward

- Continued performance optimizations as part of HPX+ efforts
  - More efficient parallelized probing and progress
  - Better control of task/thread affinity and placement (KNL systems)
  - Ability to form partitioned communication domains
  - Atomics and vectored operations
- Additional Photon backends being developed
  - Zero-copy shared memory
  - Native GPU support (e.g. GPUDirect)
  - Adapting to evolving libfabric interface (OmniPath and KNL optimizations)
  - Optimize caching and buffer management at scale
- Continued work on network-assisted address translation
  - NetFPGAs

Photon in HPX-5 demonstrates improved performance for application with a modular design to support future generation networks

Dynamic Load balancing of FMM in HPX-5

AGAS in HPX-5 enables asynchronous load balancing in FMM through active migration of nodes in the global spatial decomposition tree. N-body like problems appear in many scientific applications. The naïve solution has $O(N^2)$ complexity, whereas the Fast Multipole Method(FMM) reduces this to O(N) complexity up to any prescribed accuracy requirement. During the load-balancing phase, the optimal data distribution is determined by performing edge-cut recursive partitioning of the aggregated communication graph.

- In HPX-5, load balancing naturally takes the form of global data relocation
- Lightweight online recording of global block statistics.
- Edge-cut, multi-level recursive partitioning of the block graph.
- Blocks remapped/moved across localities concurrently.

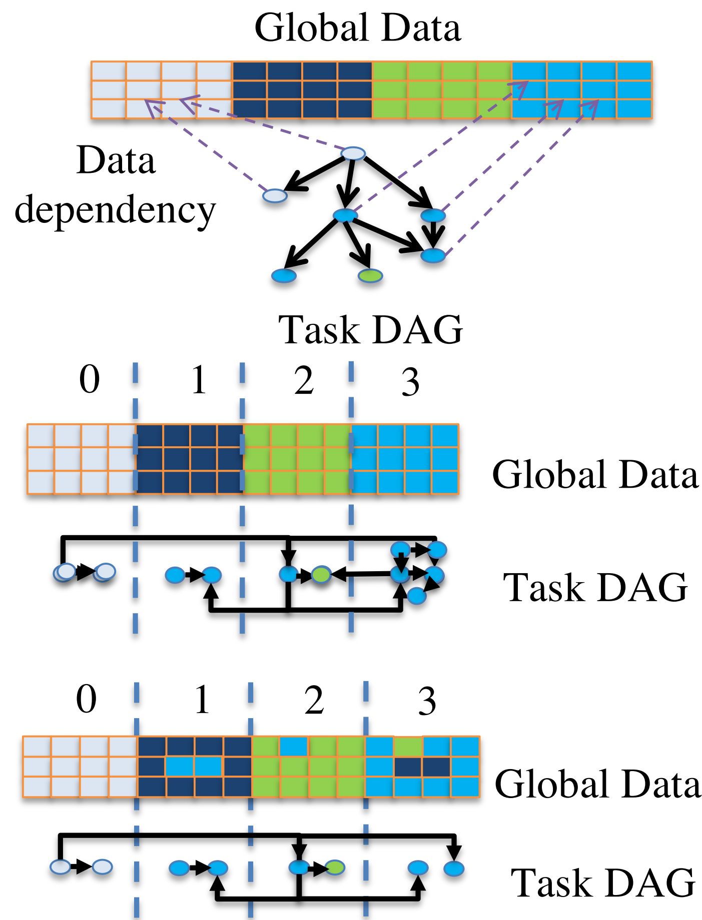


Figure 11: Dynamic Load Balancing in AGAS - Lightweight tasks operating in parallel on global data distributed across different nodes and work balancing (through stealing or sharing)

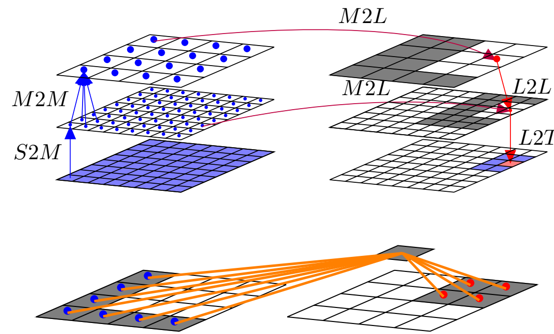


Figure 12: DAG representing the fusion of two spatial partitions with source ensemble, target ensemble and bipartite graphs connecting the two

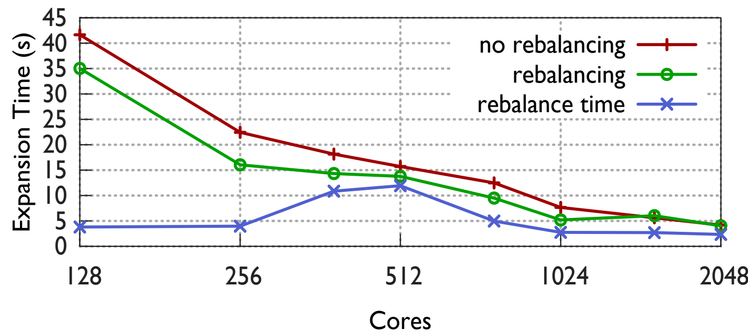


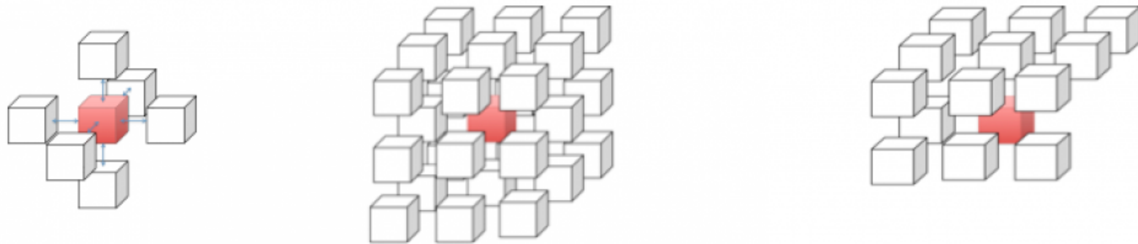
Figure 13: Performance results on Cori NERSC - Cray XC40 supercomputer with two 16-cores Intel "Haswell" processor at 2.3 GHz and Cray Aries with Dragonfly topology.

LULESH

Proxy applications representing kernels of scientific computation toolkits have been created and implemented in multiple programming models by the Department of Energy's co-design centers, in part, to directly compare performance characteristics and to explore and quantify their benefits. One of these proxy applications is LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [2], which has already played a key role in comparing and contrasting programmability and performance among key programming models. In Karlin et al.[3], a comparison of LULESH among OpenMP, MPI, MPI+OpenMP, CUDA, Chapel, Charm++, Liszt, and Loci was presented with a focus on programmer productivity, performance, and ease of optimizations. HPX-5 has also implemented LULESH help further comparisons between these programming models.

The LULESH algorithm simulates the Sedov blast wave problem in three dimensions. The problem solution is spherically- symmetric and the code solves the problem in a parallelepiped region. The LULESH algorithm is implemented as a hexahedral mesh-based code with two centerings. Element centering stores thermodynamic variables such as energy and pressure. Nodal centering stores kinematics values such as positions and velocities. The simulation is run via time integration using a Lagrange leapfrog algorithm. There are three main computational phases within each time step: advance node quantities, advance element quantities, and

calculate time constraints. There are three communication patterns, each regular, static, and uniform: face adjacent, 26 neighbor, and 13 neighbor communications, illustrated below:



As a regular, uniform, and static mini-application, LULESH is well suited for a programming model like MPI, showing very few inefficiencies in computational phase diagrams such as the following, obtained using vampirtrace:

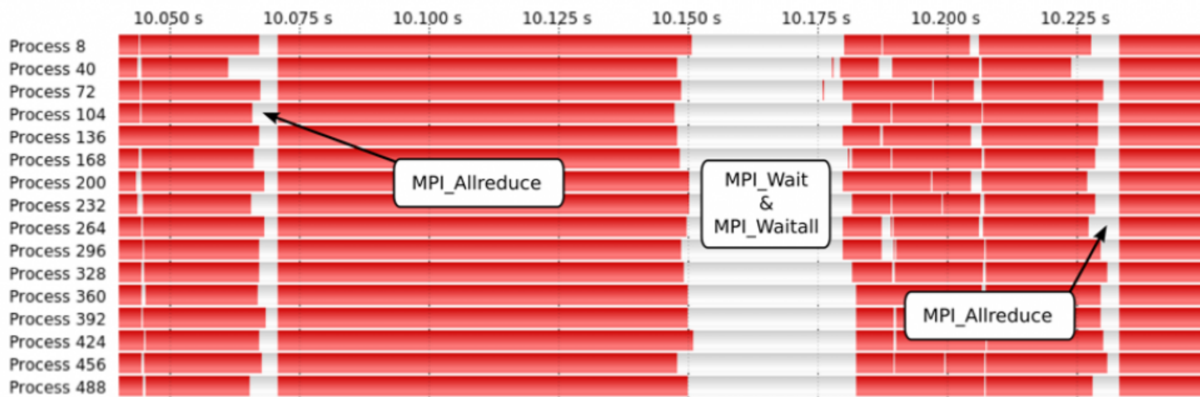


Figure 14: In this figure, white phases indicate waiting for communication while red phases indicate computation. LULESH computational phases are dominated by computation.

The HPX-5 version of LULESH comes in two flavors. The first, known as the parcels implementation, does not take advantage of parallel for loop opportunities and is more directly comparable to the MPI only implementation of LULESH. The other flavor, known as the omp parcels version, does take advantage of parallel for loops and is more comparable to the MPI-OpenMP implementation of LULESH. Both use command line arguments to control the behavior of the mini-application. The number of domains, controlled by the -n option, controls how many domains of local grids will be simulated. This number must be a perfect cube. The -x option indicates the number of points cubed in each domain. The omp parcels version is intended to be run where there is only one domain per locality, thereby necessitating that the local grid size be scaled appropriately by the number of cores per locality. More specific instructions can be found by invoking help on the application executable.

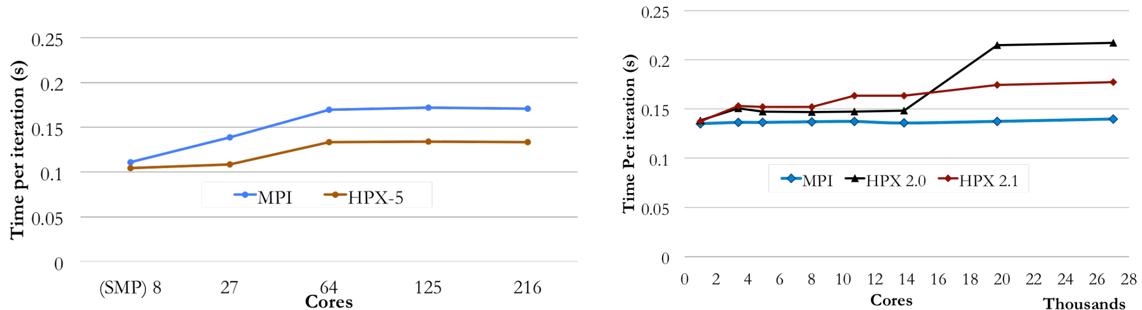


Figure 15: Large core count, weak scaling tests of LULESH conducted on Cray XC30 including MPI and two HPX-5 implementations featuring different communication mechanisms.

Weak scaling simulations of the HPX-5 implementation of LULESH were conducted across a wide range of core counts and results were compared with the reference MPI implementation (Figure 15). Simulations were performed on NERSC's Edison, a Cray XC30 platform using the Aries interconnect and Intel Xeon processors. LULESH was run for 500 iterations with a local problem size of 48^3 on each core. For each data point in the weak scaling tests, results were averaged over five runs.

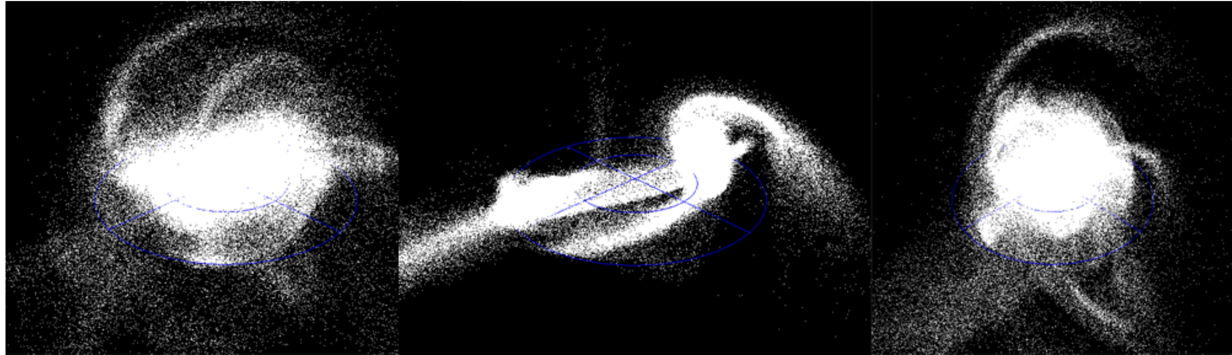
As the results show, the best HPX-5 LULESH performance is still slower than the reference MPI implementation. One of the reasons is that LULESH is a regular, static, and uniform code that is naturally portable to MPI model of computation. Secondly, the HPX-5 port does not introduce significant changes to the LULESH algorithm implemented by the MPI variant, opting to emulate the original message patterns using point-to-point communication mode rather than native, asynchronous, one-sided messages. The additional inefficiencies introduced by the emulation layer become dominant when the benchmark scale grows sufficiently. Finally, modern MPI libraries provide highly optimized implementations of collective calls, featuring tens of possible code paths and algorithms that are dynamically selected based on network status, topology of communication, and application layout. Replicating this knowledge in a different communication environment over limited time is extremely challenging and presents one of the critical targets for future HPX development.

DASHMM

The Dynamic Adaptive System for Hierarchical Multipole Methods [4] is an easy-to-use, extensible library implementing multipole methods. DASHMM uses a dynamic adaptive runtime system (HPX-5) to improve the efficiency and scalability of multipole method calculations. DASHMM is extendable through user-defined methods and expansions allowing a domain scientist to implement the details of their particular problem, while allowing the library to handle the effective parallelization of the computation.

The current version of DASHMM (1.2.0) operates on both shared and distributed memory architectures. It includes implementations of three multipole methods: Barnes-Hut, classical Fast Multipole Method (FMM), a variant of FMM that uses exponential expansions (referred to as FMM97), and permits easy extension to other multipole or multipole-like methods. It

provides built-in Laplace, Yukawa and Helmholtz kernels, covering a wide range of application use-cases. DASHMM also comes with three demonstration codes that exercise the basic interface, demonstrates how DASHMM can be used in a time-stepping scheme, and how a user might register their own expansion with the library.



Shown below is a strong scaling test on the Cori supercomputer. The library was tested for all three kernels using the FMM97 method (Laplace in blue, Yukawa in yellow, and Helmholtz in red) for two different source and target distributions (inside the volume of a cube with square marks, and on the surface of a sphere with circular marks).

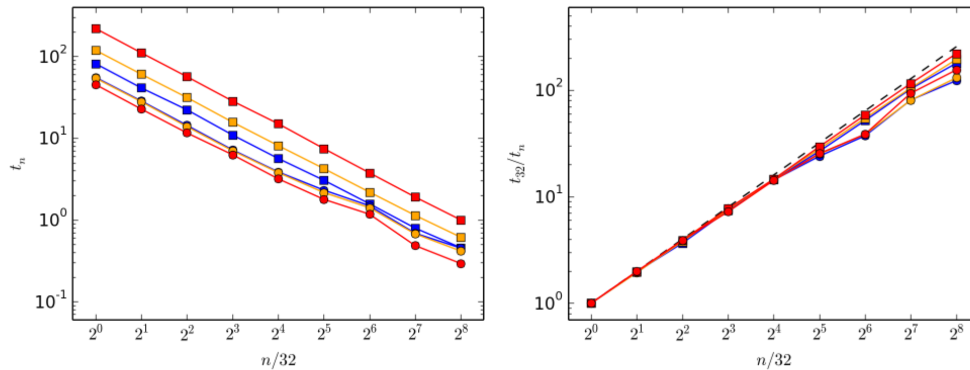


Figure 16: The left panel shows the absolute time for the evaluation and the right panel shows the speedup relative to one node.

libPXGL

Following the philosophy of design of Standard Template Library (STL) and the trail of evolution of Parallel Boost graph library (PBGL), libPXGL is the beginning of an endeavor to develop a next-generation graph library based on HPX-5.

Our first effort is invested in implementing graph algorithms for solving the Single Source Shortest Path (SSSP) problem as SSSP is a good representative of a class of irregular graph problems. Given a graph and a source, the SSSP problem asks to find out the shortest possible distances from the source to all other vertices in the graph. For example, the graph in the figure (a) has five vertices s, t, x, y, and z. The distances between adjacent vertices are shown on the directed edges. Let us choose vertex s as the source. Figure(b) shows the shortest path from s to all other vertices.

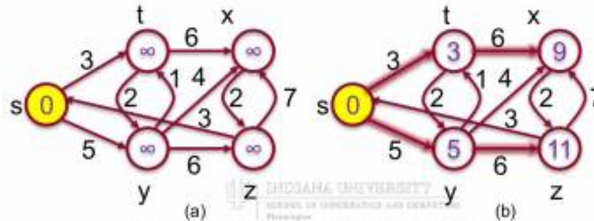
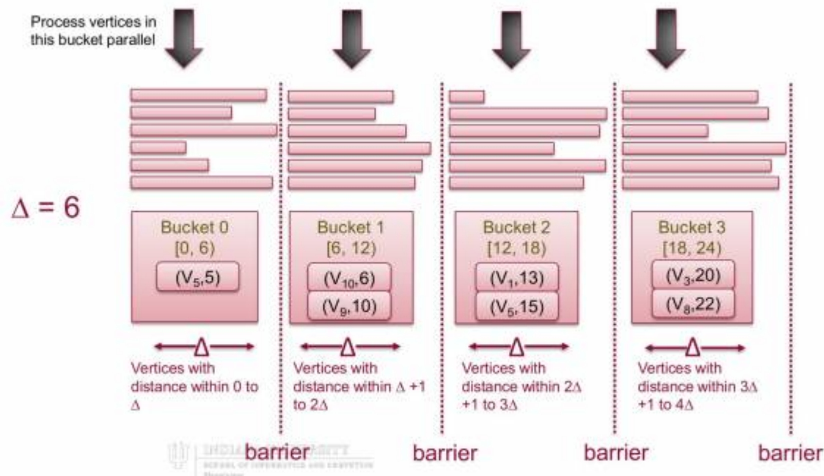


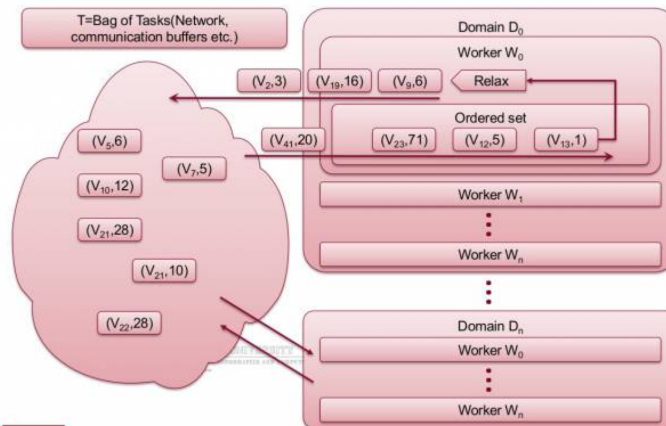
Figure: (a) A weighted, directed graph
(b) Shortest path tree rooted at s

Four graph algorithms are implemented in HPX. In particular, we have implemented chaotic, Delta-stepping, Distributed control and k-level asynchronous algorithms. In what follows, we give a brief description of how each algorithm works.

In the **chaotic** algorithm, the distances from the source are updated asynchronously without imposing any order. There is no barrier requirement for algorithm progress and the distances are guaranteed to reach the minimum value at the end of the algorithm. In the **Delta-stepping** algorithm, the distance space is divided into several intervals. Each interval is treated as a bucket. The algorithm proceeds by processing all vertices in each bucket without imposing any order. Once processing is done for the current bucket, the algorithm can progress to the next bucket. Between each bucket processing, there is an explicit barrier on which all localities need to wait before proceeding to the next bucket. The following figure is a high level schematic of the delta-stepping algorithm.



Distributed Control is an approach to achieve optimistic parallelism by removing any requirement for synchronization. In doing so, it also thrives for local ordering based on thread-local priority queues on each locality to minimize redundant work done. The following figure is a high level schematic of distributed control algorithm.



The fourth algorithm is based on the **k-level asynchronous** paradigm. In this approach, the algorithm progresses k-level wise. Within each k-level, the algorithm can progress asynchronously. For example if k=2, then in the first step, the algorithm will process all the vertices within the reach of the source by at most 2 steps without imposing any order. The next step will process all the vertices reachable with atmost 4 steps and so on.

The algorithms for SSSP can be run with the 9th DIMACS challenge input graphs and problem specification files. The DIMACS distribution contains the generators and the makefiles necessary for generating all the DIMACS input files. Additionally, the algorithms can also be run with Graph 500 input graphs. There is an implementation of Graph 500 specification-based graph generator included in HPX-5 to generate the graphs at different scale as well as the problem instances.

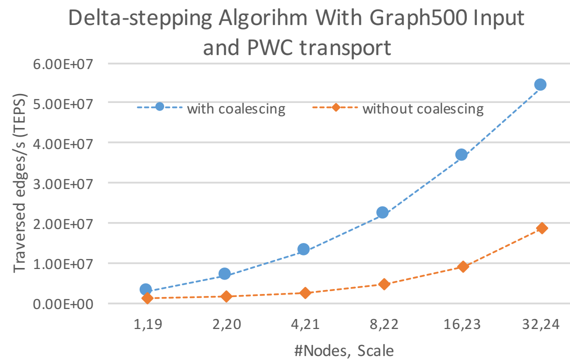


Figure 17: Performance of Delta-Stepping algorithm for Single-Source Shortest Paths (SSSP) executed on Graph500 benchmark inputs with and without coalescing

HPX-5 provides efficient intra-node and inter-node parallelism with load balancing and overlap of computation and communications. Furthermore, HPX-5 is getting more efficient at this task with every version, in part due to careful co-design with graph analytics applications such as single-source shortest paths (SSSP) as shown in the figure. We have been investigating how to attune the HPX-5 runtime to the needs and patterns of irregular applications. Traditional applications have some global flow control that relies on globally synchronized data structures. However, data-driven problems can rely on local flow control, but such approach requires careful consideration of the runtime scheduler, which can greatly degrade the performance of such data-driven applications. Data-driven graph-like applications are an important and emerging class of problems. HPX-5 is particularly well suited for execution of such problems at exascale.

Wavelet Adaptive Multiresolution Representation (Wavelet AMR)

While the wavelet transformation has historically figured prominently in compression algorithms, it has become a crucial tool in solving partial differential equations. Hierarchical adaptive wavelet is well suited for solving partial differential equations where localized structures develop intermittently in the computational domain. The advantage of using wavelets in this is that it requires far fewer collocation points than other comparable algorithms while the wavelet amplitude provide a direct measure of the local approximation error at each point, thereby providing a built-in refinement criterion. The HPX-5 implementation of the Wavelet-AMR spawn threads for each collocation point while managing the sparse grid of collocation points inside the global address space. There are three systems of equations implemented in Wavelet-AMR: The Einstein Equations, the relativistic magneto hydrodynamics equations, and the Euler equations. Its communication patterns are irregular, dynamic, and non uniform. This application comes in three flavors: serial, cilk, and HPX-5.

To appreciate the sparse grid of collocation points used for the RMHD point blast wave evolution, the collocation points at one timestep are shown below:

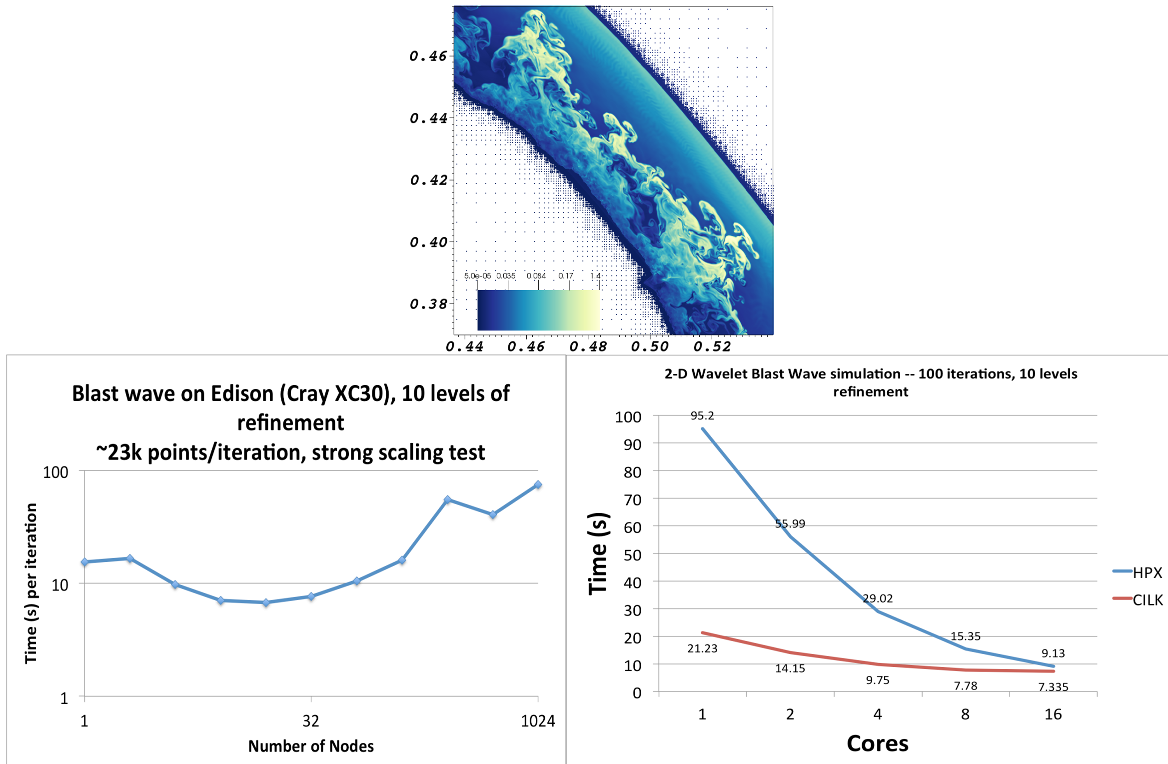


Figure 18: Distributed strong scaling adaptive simulation on Edison.

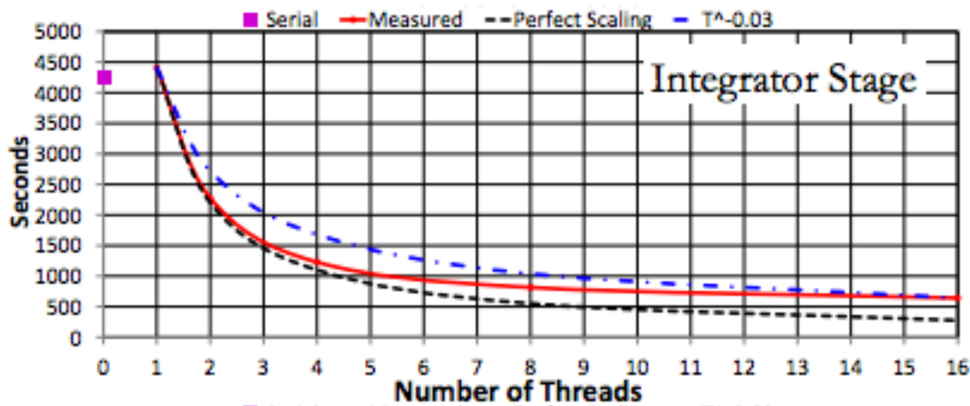


Figure 19: Asynchronous Multi-Tasking (AMT) models the on-node multithreading performance of the Wavelet AMR code

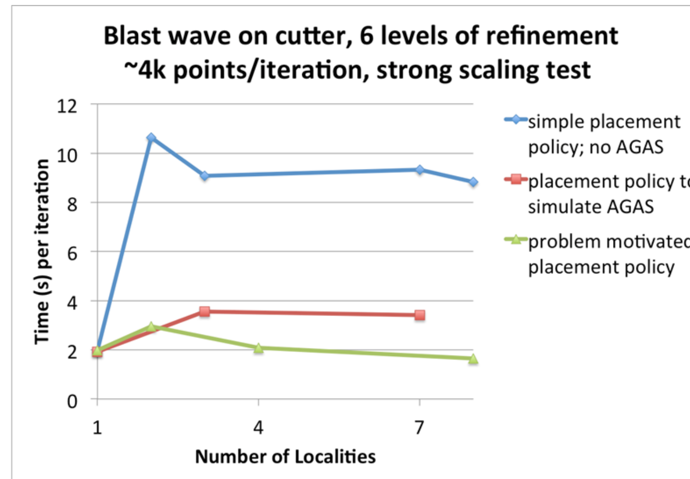


Figure 20: Explores the impact of simple movement policies for AGAS.

High Performance Conjugate Gradients (HPCG)

The HPX-5 HPCG is a comprehensive port of the MPI-OpenMP HPCG reference implementation. It closely follows the MPI-OpenMP implementation but using active messages and HPX-5 semantics. High Performance Conjugate Gradients (HPCG) is a benchmark project aimed to create a more relevant metric for ranking HPC systems. Alterations to sparse matrix-vector multiply include RDMA based put and get approaches have been added. Also added is a new demonstration of an HPX+MPI legacy support modality where HPX and MPI coexist. In this modality, some kernels use HPX whilst others use MPI in order to demonstrate a path forward for porting large legacy applications.

Its communication patterns are regular, static, and nonuniform. Includes:
- Sparse matrix-vector multiplication
- Sparse triangular solve
- Global dot products
- Multigrid preconditioned conjugate gradient

In the performance plots below Plot1 shows the performance of SpMV on small cluster with 16 cores/node. In this plot, higher the Gflops, better the results are. Two modalities are explored: one computational domain per core and over decomposition. When using one computational domain per core, both the reference MPI and parcels versions of SpMV have very similar performance. In the over decomposed modality, while both the parcels and reference MPI versions show improved performance, the parcels version substantially outperforms the reference MPI implementation.

Plot2: The results on the large scale machine show similar outcomes to that found on the small scale cluster. The parcels version without over decomposition closely tracks the reference MPI implementation performance. But by applying a simple over decomposition factor of 8, the performance of the parcels version nearly doubles. A more careful application of the same series of simulations to address NUMA concerns leads to another significant performance improvement for SpMV.

Plot3: The results on the large scale machine show similar outcomes to that found on the small scale cluster. The parcels version without over decomposition closely tracks the reference MPI implementation performance. But by applying a simple over decomposition factor of 8, the performance of the parcels version nearly doubles. A more careful application of the same series of simulations to address NUMA concerns leads to another significant performance improvement for SpMV.

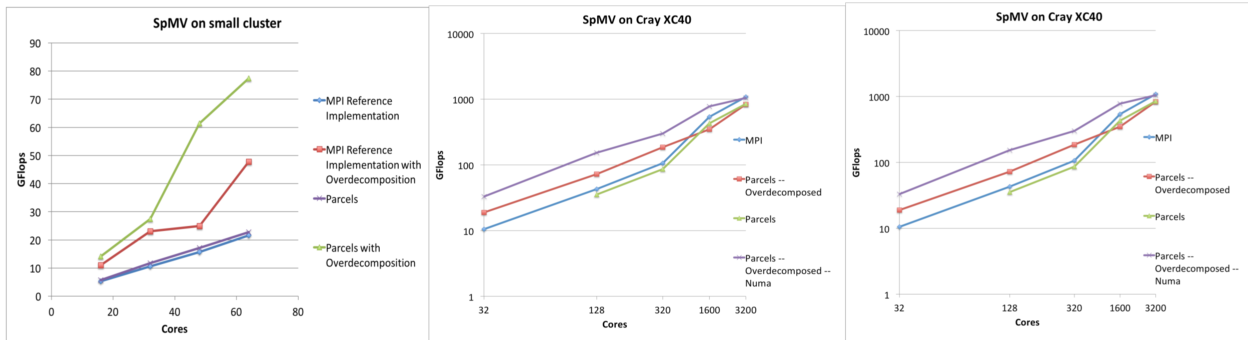


Figure 21: Performance of SpMV

A complete picture of the average GFlops achieved with spmv for the parcels and memget strategies on the large-scale machine with 32 cores/node. In this series of experiments, different algorithms have different colors. Triangles indicate 32 threads, squares indicate 16 threads, and circles indicate 2 threads. Memget experiments came in two varieties: 1x means the number of chunks in the parallel for loops matches the number of threads while 2x means that there are twice as many chunks of data in the parallel for loops as number of threads. The parcels strategy with over decomposition performs the best.

Parcels approach

```
1  rank = malloc Comm Size
2  init(rank)
3  for i from 0 to Iterations
4    for phase in Phases
5      for neighbor in Neighbors(rank, phase)
6        recvs += IRECV(rank, phase, neighbor)
7      for neighbor in Neighbors(rank, phase)
8        Pack(rank, phase, neighbor)
9        ISEND(rank, phase, neighbor)
10     WAIT_ALL(recvs)
11     unpack(rank, phase, neighbor)
12     parallel_for(computation(rank, phase))
```

RDMA memput approach

```
1  rank = malloc Comm Size
2  init(rank)
3  for i from 0 to Iterations
4    for phase in Phases
5      for neighbor in Neighbors(rank, phase)
6        memput(neighbor,lco[neighbor])
7
8      wait(lco[rank])
9      parallel_for(computation(rank, phase))
```

RDMA memget approach

```
1  rank = malloc Comm Size
2  init(rank)
3  for i from 0 to Iterations
4    for phase in Phases
5      set_lco(rank)
6      parallel_for xRow in NeighborXRows(rank, phase)
7        memget(xRow,lco[xRow])
8      parallel_for(computation(RowsOfA[rank], phase))
9      wait(xRowsLCO)
10 where
11     on access to a remote rows of x while working on a row
12     1) spawn a thread to finish computation
13     2) on access to to any remote xRow: wait(lco[xRow])
14     3) after computation is finished: set_lco(xRowsLCO)
```

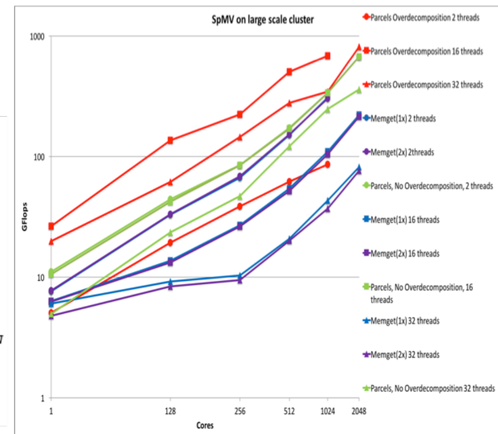


Figure 22: RDMA with asynchronous multitasking runtime (AMT) systems open up a series of different SpMV strategies with some strategies resulting in vastly superior performance depending on the AMT runtime system overhead.

Human Brain Simulation (BlueBrain Project @ EPFL) – HPX-5 for Asynchronous simulation of detailed neural networks

The goal of the Blue Brain project [5] is to build biologically detailed digital reconstructions and simulations of the rodent, and ultimately the human brain. It offers a radically new approach for understanding the multilevel structure and function of the brain. It exploits interdependencies in the experimental data to obtain dense maps of the brain, without measuring every detail of its multiple levels of organization (molecules, cells, micro-circuits, brain regions, the whole brain). Supercomputer-based simulation of their behavior turns understanding the brain into a tractable problem, providing a new tool to study the complex interactions within different levels of brain organization and to investigate the cross-level links leading from genes to cognition.

- Dynamically adaptive software to allow simulation at different scales:
  - Point neuron level simulation (thousands/millions of neurons per node)
  - Compartmental level simulation (few neurons per node)
  - Biomolecular level simulation (one neuron across several nodes)
- Multirate and variable time-step solvers (based on each different mechanism) reflect better the neuronal networks behavior, contrarily to fixed time-step solvers
  - This requires a totally asynchronous programming paradigm as provided by HPX
- Hide communication and threading complexity
  - Developer only focus on writing the logic; HPX-5 handles parallelization
- Transparent load balancing

- Task stealing queue allows balancing of work across threads
- Global Address Space allows memory to move to different localities to balance work across nodes

- Removal of collective communication and computation calls:
  - Simulation should be a *free system* where computation of objects is independent
  - Suitable for simulation of objects with unpredictable execution times

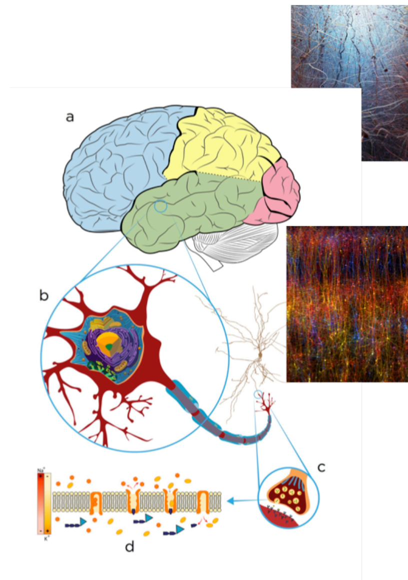


Figure 23: General use of neural network and multi-scale simulation

We use HPX-5 futures for a tree-based parallelism of a sparse-tridiagonal matrix representing the branching between neurons. No benchmark results available.

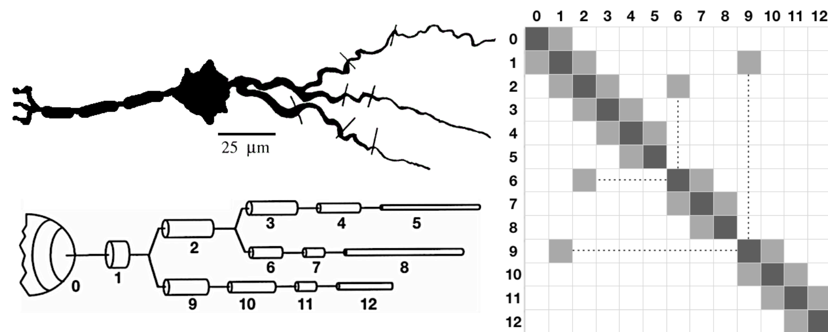


Figure 24: A scanned neuron model, a compartment-based model and its matrix representation

We use HPX-5 and-gates for graph-based parallelism of biological phenomena. Read/write dependencies are collected from parsing the DSL language specifying the equations. Benchmark results presented at the end.

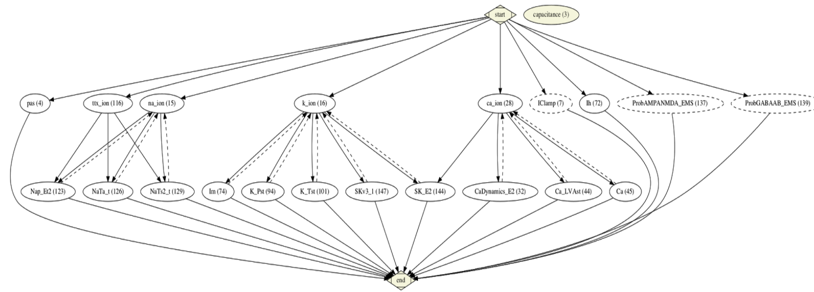


Figure 25: Graph-parallelism to a neuron driven by 22 mechanisms

We use HPX-5 to investigate new stepping policies: a) synchronous barrier for fixed communication barrier, b) asynchronous all-reduce for sliding time window and c) conditional barriers for time-dependencies based on the synaptic delay between pairs of neurons.

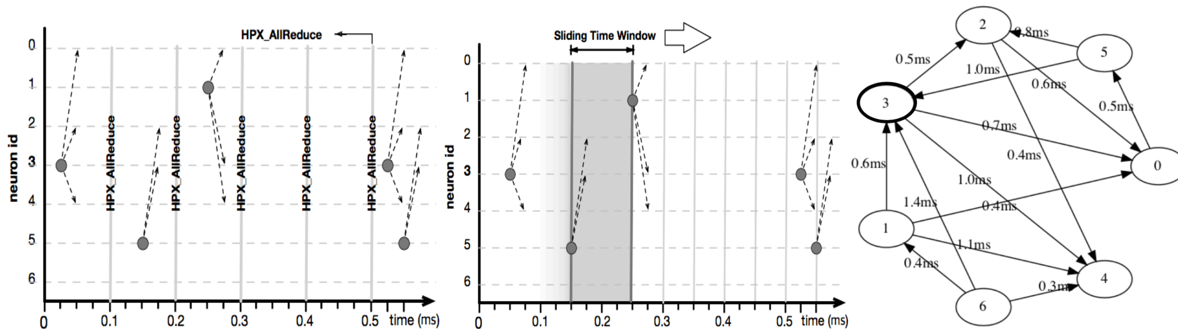
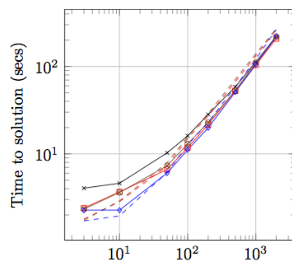


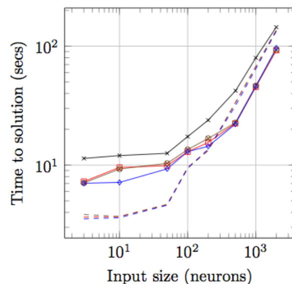
Figure 26: Schematic behavior of the three algorithms

We present benchmarks for the three aforementioned stepping algorithms. Results are presented with and without mechanisms graph-parallelism.



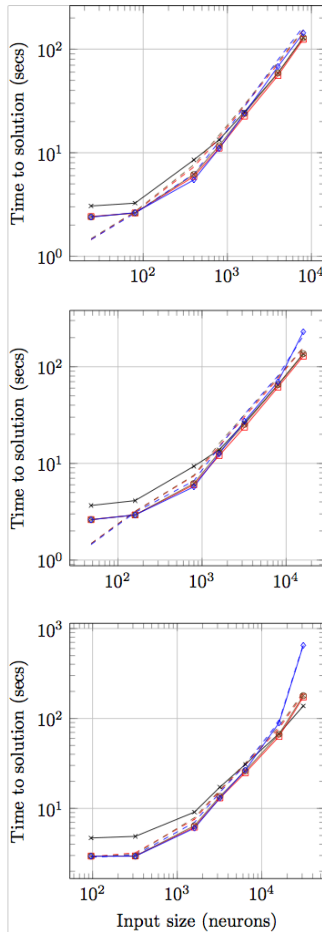
Intel Sandy Bridge E5-2670 (16 cores @ 2.6GHz), 1 compute node, gcc 4.9.3 compiler

total number of neurons	3	10	50	100	200	500	1000	2000
Reference Solution	4.05	4.61	10.27	16.08	28.26	58.53	112.54	220.73
Asynchronous All-Reduce	**2.40**	**3.67**	6.63	11.78	21.64	52.57	103.87	208.23
Async. All-Reduce + Mechs-graph	1.79	2.84	7.30	15.05	27.07	66.70	132.50	263.87
Sliding Time Window	**2.33**	**3.62**	7.31	13.01	22.82	52.91	110.67	218.36
Sliding Time Window+ Mechs-graph	1.74	2.90	7.93	15.73	28.93	71.12	139.91	271.82
Time-Dependency LCO	2.27	2.27	6.03	11.11	19.84	50.89	105.33	222.58
Time-Dependency + Mechs-graph	1.71	1.95	6.15	12.69	23.00	58.48	121.40	265.02
Best-case speed-up	2.37x	2.36x	1.70x	1.48x	1.42x	1.15x	1.08x	1.06x



Intel KNL, Xeon Phi (64 cores, 128 threads @ 1.3GHz), 1 compute node, Intel icc17 compiler

total number of neurons	3	10	50	100	200	500	1000	2000
Reference Solution	11.38	12.02	12.56	17.35	23.85	42.19	79.74	144.49
Asynchronous All-Reduce	**7.28**	**9.56**	9.88	12.92	15.50	22.39	45.46	92.28
Async. All-Reduce + Mechs-graph	3.65	3.69	4.62	9.54	13.36	33.35	67.13	93.39
Sliding Time Window	**7.08**	**9.27**	10.34	13.50	16.75	22.67	46.46	93.39
Sliding Time Window+ Mechs-graph	3.84	3.70	4.68	9.478	13.21	33.38	67.37	136.14
Time-Dependency LCO	7.03	7.15	9.29	12.98	14.39	22.11	45.92	96.60
Time-Dependency + Mechs-graph	3.53	3.61	4.60	9.40	13.60	31.36	64.43	133.80
Best-case speed-up	3.22x	3.33x	2.73x	1.85x	1.81x	1.88x	1.75x	1.57x



Intel Sandy Bridge E5-2670 (16 cores @ 2.6GHz), 8 compute nodes, mvapich2 compiler

total number of neurons	24	80	400	800	1600	4000	8000
number of neurons per compute node	3	10	50	100	200	500	1000
Reference Solution	3.06	3.26	8.52	13.33	24.79	58.80	128.76
Asynchronous All-Reduce	2.42	2.62	5.90	11.08	22.52	55.67	124.31
Async. All-Reduce + Mechs-graph	1.45	2.68	7.26	14.05	28.42	71.83	152.62
Sliding Time Window	2.40	2.62	6.16	11.72	23.87	58.86	129.31
Sliding Time Window+ Mechs-graph	1.48	2.70	7.64	14.79	29.17	75.00	153.52
Time-Dependency LCO	2.39	2.64	5.47	10.98	24.01	67.93	143.21
Time-Dependency + Mechs-graph	1.45	2.61	6.25	12.86	27.79	78.99	164.12
Best-case speed-up	3.22x	3.33x	2.73x	1.85x	1.81x	1.88x	1.75x

Intel Sandy Bridge E5-2670 (16 cores @ 2.6GHz), 16 compute nodes, mvapich2 compiler

total number of neurons	48	160	800	1600	3200	8000	16000
number of neurons per compute node	3	10	50	100	200	500	1000
Reference Solution	3.67	4.12	9.33	13.81	26.23	64.89	135.47
Asynchronous All-Reduce	2.62	2.93	6.01	12.06	23.67	61.47	128.31
Async. All-Reduce + Mechs-graph	1.48	3.19	7.40	15.07	30.25	75.58	157.11
Sliding Time Window	2.62	2.93	6.20	13.03	25.30	65.06	136.04
Sliding Time Window+ Mechs-graph	1.48	3.18	7.53	16.02	31.86	78.34	158.04
Time-Dependency LCO	2.61	2.93	5.72	12.55	27.31	69.89	229.85
Time-Dependency + Mechs-graph	1.45	3.11	6.60	14.65	32.18	80.40	208.46
Best-case speed-up	3.22x	3.33x	2.73x	1.85x	1.81x	1.88x	1.75x

Intel Sandy Bridge E5-2670 (16 cores @ 2.6GHz), 32 compute nodes, mvapich2 compiler

total number of neurons	96	320	1600	3200	6400	16000	31000
number of neurons per compute node	3	10	50	100	200	500	969
Reference Solution	4.69	4.89	9.10	17.25	30.88	67.74	137.72
Asynchronous All-Reduce	2.95	2.96	6.18	13.08	24.90	63.00	172.96
Async. All-Reduce + Mechs-graph	2.91	3.21	7.56	15.75	30.61	77.39	188.33
Sliding Time Window	2.95	2.95	6.42	13.65	26.12	66.79	179.41
Sliding Time Window+ Mechs-graph	2.90	3.11	7.87	16.06	31.97	81.57	195.21
Time-Dependency LCO	2.94	2.94	6.03	13.04	26.70	89.15	650.21
Time-Dependency + Mechs-graph	2.86	3.03	6.99	15.08	30.67	92.43	662.34
Best-case speed-up	3.22x	3.33x	2.73x	1.85x	1.81x	1.88x	1.75x

Figure 27: Results of stepping algorithms

Publications

[1] Thomas Sterling, Daniel Kogler, Matthew Anderson, and Maciej Brodowicz. **SLOWER: A performance model for Exascale computing**. *Supercomputing Frontiers and Innovations*, 1:42–57, September 2014.

[2] Abhishek Kulkarni, Luke Dalessandro, Ezra Kissel, Andrew Lumsdaine, Thomas Sterling, and Martin Swany. **Network-Managed Virtual Global Address Space for Message-driven Runtimes**. In *Proceedings of the 25th International Symposium on High Performance Parallel and Distributed Computing (HPDC 2016)*, June 2016.

[3] Ezra Kissel and Martin Swany. **Photon: Remote Memory Access Middleware for High-Performance Runtime Systems**. In *In Proceedings of the 1st Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM) Workshop (Co-located with IPDPS)*, May 2016.

[4] Matthew Anderson, Maciej Brodowicz, Luke Dalessandro, Jackson DeBuhr, and Thomas Sterling. **A dynamic execution model applied to distributed collision detection**. In *29th International Supercomputing Conference (ISC), 2014*, Leipzig, Germany, June 2014.

[5] Thomas Sterling, Matthew Anderson, P. Kevin Bohan, Maciej Brodowicz, Abhishek Kulkarni, and Bo Zhang. **Towards Exascale Co-design in a Runtime System**. In *Exascale Applications and Software Conference*, Stockholm, Sweden, April 2014.
[6] Alice Koniges, Jayashree Ajay Candadai, Hartmut Kaiser, Adrian Serio, Kevin Huck, Jeremy Kemp, Thomas Heller, Friedrich-Alexander, Matthew Anderson, Andrew Lumsdaine, Thomas Sterling, and Ron Brightwell. **HPX Applications and Performance Adaptation**, Poster Presentation at the SC15 Posters, SC15 Austin, Texas.

References

[1] HPX: http://hpx.crest.iu.edu
[2] Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH);. Available from: https://codesign.llnl.gov/lulesh.php
[3] Karlin I, Bhatele A, Keasler J, Chamberlain BL, Cohen J, DeVito Z, et al. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In: Proc. of the 27-th IEEE International Parallel and Distributed Processing Symposium (IPDPS); 2013.
[4] DASHMM: https://www.crest.iu.edu/projects/dashmm/.
[5] BlueBrain: http://bluebrain.epfl.ch