An Autonomic Performance Environment for Exascale

Kevin A. Huck\textsuperscript{1}, Nick Chaimov\textsuperscript{1}, Allen D. Malony\textsuperscript{1}, Sameer Shende\textsuperscript{1}, Allan Porterfield\textsuperscript{2}, Rob Fowler\textsuperscript{2}

\textsuperscript{1}University of Oregon, Eugene, Oregon, USA
\textsuperscript{2}RENCI, Chapel Hill, North Carolina, USA
What I am going to tell you...

• XPRESS project overview
  – OpenX, ParalleX, HPX etc.
• Asynchronous tasking runtimes, tool challenges
• APEX overview
• Examples
**XPRESS**

- DOE X-Stack project to develop the OpenX software stack, implementing the ParalleX execution model
  - Sandia National Laboratories (R. Brightwell)
  - Indiana University (T. Sterling, A. Lumsdane)
  - Lawrence Berkeley National Laboratory (A. Koniges)
  - Louisiana State University (H. Kaiser)
  - Oak Ridge National Laboratory (C. Baker)
  - University of Houston (B. Chapman)
  - University of North Carolina (A. Porterfield, R. Fowler)
  - University of Oregon (A. Malony, K. Huck, S. Shende)

OpenX architecture

The two major components of the XPRESS OpenX software architecture are the LXK operating system and the HPX runtime system software. Between these two is a major interface protocol, the “Prime Medium” that supports bidirectional complex interoperability between the operating system and the runtime system. The interrelationship is mutually supporting, bidirectional, and dynamically adaptive.

Figure 1 shows key elements of the runtime system including lightweight user multithreading, parcel message-driven computation, LCO local control objects for sophisticated synchronization, and AGAS active global address space and processes. Each application has its own ephemeral instance of the HPX runtime system. The figure also shows the LXK operating system on the other side of the Prime Medium consisting of a large ensemble of persistent lightweight kernel supervisors each dedicated to a particular node of processor, memory, and network resources. It is the innovation and power of the OpenX software architecture that the runtime system and the operating system employ each other for services. Both are simple in design but achieve complexity of operation through a combination of high replication and dynamic interaction.

XPI, the low-level imperative API, will serve both as a readable interface for system software development and early experimental application programming with which to conduct experiments. It will also serve as a target for source-to-source compiler translation from high-level APIs and languages. XPI will represent the ParalleX semantics as closely as possible in a familiar form: the binding of calls to the C programming language. Part of the compiler challenge is to bridge XPI to the HPX runtime system.

Domain specific programming will be provided through a metaprogramming framework that will permit rapid development of DSLs for diverse disciplines. An example of one such DSL derived from the metatoolkit will be developed by the XPRESS project. Finally, targeting either XPI or native calls of the HPX runtime will be compilation strategies and systems to translate MPI and OpenMP legacy codes to a form that can be run by OpenX with performance at least as good as a native code implementation.
ParalleX, HPX, and OpenX

• ParalleX is an experimental execution model
  – Theoretical foundation of task-based parallelism
  – Run in the context of distributed computing context
  – Supports integration of the entire system stack

• HPX is the runtime system that implements ParalleX
  – Exposes an uniform, standards-oriented API
  – Enables writing a fully asynchronous code using hundreds of millions of threads
  – Provides unified syntax and semantics for local and remote operations
    – HPX-3 (LSU, C++ language/Boost)
    – HPX-5 (IU, C language)

• XPRESS will develop an OpenX exascale software stack based on ParalleX
ParalleX / HPX

- Governing principles (components)
  - Active global address space (AGAS) instead of PGAS
    - Enables seamless distributed load balancing and adaptive data placement and locality control
  - Message-driven via Parcels instead of message passing
    - Enables transparent latency hiding and event driven computation
    - Parcels are a form of active messages
  - Lightweight control objects (LCO) instead of global barriers
    - Reduces global contention and improves parallel efficiency
    - Synchronization semantics
  - Moving work to data instead of moving data to work
    - Basis for asynchronous parallelism and dataflow style computation
  - Fine-grained parallelism of lightweight Threads (vs. CSP)
    - Improves system and resource utilization
An Autonomic Performance Environment for Exascale
HPX-5 Architecture
OpenX Software Stack and APEX

The two major components of the XPRESS OpenX software architecture are the LXK operating system and the HPX runtime system. Between these two is a major interface protocol—the “Prime Medium” that supports bidirectional complex interoperability between the operating system and the runtime system. The interrelationship is mutually supporting, bidirectional, and dynamically adaptive.

Figure 1 shows key elements of the runtime system including lightweight user multithreading, parcel message-driven computation, LCO local control objects for sophisticated synchronization, and AGAS active global address space and processes. Each application has its own ephemeral instance of the HPX runtime system. The figure also shows the LXK operating system on the other side of the Prime Medium consisting of a large ensemble of persistent lightweight kernel supervisors each dedicated to a particular node of processor, memory, and network resources. It is the innovation and power of the OpenX software architecture that the runtime system and the operating system employ each other for services. Both are simple in design but achieve complexity of operation through a combination of high replication and dynamic interaction.

XPI, the low-level imperative API, will serve both as a readable interface for system software development and early experimental application programming with which to conduct experiments. It will also serve as a target for source-to-source compiler translation from high-level APIs and languages. XPI will represent the ParalleX semantics as closely as possible in a familiar form: the binding of calls to the C programming language. Part of the compiler challenge is to bridge XPI to the HPX runtime system.

Domain specific programming will be provided through a metaprogramming framework that will permit rapid development of DSLs for diverse disciplines. An example of one such DSL derived from the metatoolkit will be developed by the XPRESS project. Finally, targeting either XPI or native calls of the HPX runtime will be compilation strategies and systems to translate MPI and OpenMP legacy codes to a form that can be run by OpenX with performance at least as good as a native code implementation.

* From XPRESS proposal

An Autonomic Performance Environment for Exascale
Runtime Adaptation Motivation

• Controlling concurrency
  – Energy efficiency
  – Performance

• Parametric variability
  – Granularity for this machine / dataset?

• Load Balancing
  – When to perform AGAS migration?

• Parallel Algorithms (for_each…)
  – Separate what from how

• Address the “SLOWER” performance model
Tool Challenges

• Runtime wants to manage pre-emption
  – Task starts on one OS thread, at some “waiting” point is de-scheduled by the task manager (changes the stack), resumes on another (thread/core/node)
  – Where/how to instrument?

• Overhead concerns
  – ParalleX goal: million-way concurrency
  – NOTHING* shared between threads

• Need access to profile for decision control

• 2 implementations of HPX
APEX and Autonomics

• Performance awareness and performance adaptation
• Top down and bottom up performance mapping / feedback
  – Make node-wide resource utilization data and analysis, energy consumption, and health information available in real time
  – Associate performance state with policy for feedback control
• APEX introspection
  – OS (LXK) track system resource assignment, utilization, job contention, overhead
  – Runtime (HPX) track threads, queues, concurrency, remote operations, parcels, memory management
  – ParalleX, DSLs and legacy codes allow language-level performance semantics to be measured
APEX Information Flow

APEX Introspection
- Synchronous
- Asynchronous

APEX State
- Triggered
- Periodic

APEX Policy Engine

RCR Toolkit

Application

HPX

events

meta-events

actuators
Leverage runtime (HPX) to provide global introspection, state, and policies.
APEX Introspection

• APEX collects data through “inspectors”
  – *Synchronous* uses an event API and event “listeners”
    • Initialize, terminate, new thread – added to HPX runtime
    • Timer start, stop, yield*, resume* - added to HPX task scheduler
    • Sampled value (counters from HPX-5, HPX-3)
    • Custom events (meta-events)
  – *Asynchronous* do not rely on events, but occur periodically
• APEX exploits access to performance data from
  lower stack components
  – Reading from the RCR blackboard (i.e., power, energy)
  – “Health” data through other interfaces (/proc/stat, cpuinfo, meminfo, net/dev, self/status, lm_sensors, power*, etc.)
RCR: Resource Centric Reflection

- Performance introspection across layers to enable dynamic, adaptive operation and decision control
- Extends previous work on building decision support instrumentation (*RCRToolkit*) for introspective adaptive scheduling
- Daemon monitors shared, non-core resources
- Real-time analysis, raw/processed data published to shared memory region, clients subscribe
- Utilized at lower levels of the OpenX stack
- APEX introspection and policy components access and evaluate RCR values
APEX Event Listeners

• Profiling listener
  – Start event: input name/address, get timestamp, return profiler handle
  – Stop event: get timestamp, put profiler object in a queue for back-end processing, return
  – Sample event: put the name & value in the queue
  – Asynchronous consumer thread: process profiler objects and samples to build statistical profile (in HPX-3, processed/scheduled as a thread/task)

• Concurrency listener (postmortem analysis)
  – Start event: push timer ID on stack
  – Stop event: pop timer ID off stack
  – Asynchronous consumer thread: periodically log current timer for each thread, output report at termination
APEX Policy Listener

• Policies are rules that decide on outcomes based on observed state
  – *Triggered* policies are invoked by introspection API events
  – *Periodic* policies are run periodically on asynchronous thread

• Policies are registered with the Policy Engine
  – Applications, runtimes, and/or OS register callback functions

• Callback functions define the policy rules
  – “If x < y then…”

• Enables runtime adaptation using introspection data
  – Engages actuators across stack layers
  – Could also be used to involve online auto-tuning support*
APEX Global View

• All APEX introspection is collected locally
  – However, it is not limited to a single-node view
• Global view of introspection data and interactions
  – Take advantage of the distributed runtime support
    • HPX3, HPX5, MPI, …
• API provided for back-end implementations
  – *apex_global_get_value()* – each node gets data to be reduced, optional AGAS/RDMA put (push model)
  – *apex_global_reduce()* – optional AGAS/RDMA get (pull model), node data is aggregated at root node, optional broadcast back out
• Extends global view for policies
APEX Examples

- HPX-3 1-D stencil code
- HPX-5 LULESH kernel
- Experiments conducted on Edison
  - Cray XC30 @ NERSC.gov
  - 5576 nodes with two 12-core Intel "Ivy Bridge" processors at 2.4 GHz
  - 48 threads per node (24 physical cores w/ hyperthreading)
  - Cray Aries interconnect with Dragonfly topology with 23.7 TB/s global bandwidth
Concurrency Throttling for Performance

- Heat diffusion
- 1D stencil code
- Data array partitioned into chunks
- 1 node with no hyperthreading
- Performance increases to a point with increasing worker threads, then decreases
Concurrency Throttling for Performance

• Region of maximum performance correlates with thread queue length runtime performance counter
  – Represents # tasks currently waiting to execute
• Could do introspection on this to control concurrency throttling policy (*work in progress)
1d_stencil Baseline

48 worker threads (with hyperthreading)
- Actual concurrency much lower
  - Implementation is memory bound
- Large variation in concurrency over time
  - Tasks waiting on prior tasks to complete

Where calculation takes place

Event-generated metrics

An Autonomic Performance Environment for Exascale
1d_stencil w/ optimal # of Threads

- 12 worker threads
- Greater proportion of threads kept busy
  - Less interference between active threads and threads waiting for memory
- Much faster
  - 61 sec. vs 138 sec.
1d_stencil Adaptation with APEX

- Initially 48 worker threads
- Discrete hill climbing search to minimize average number of pending tasks
- Converges on 13 (vs. optimal of 12)
- Nearly as fast as optimal
  - 64 seconds vs. 61 seconds
1D stencil – adapting grain size

An Autonomic Performance Environment for Exascale
OpenMP version, adapt concurrency and block size with ActiveHarmony policy
Throttling for Power

• Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)
  – Proxy application in DOE co-design efforts for exascale
  – CPU bounded in most implementations (use HPX-5)

• Develop an APEX policy for power
  – Threads are idled in HPX to keep node under power cap
  – Use hysteresis of last 3 observations
  – If Power < low cap increase thread cap
  – If Power > high cap decrease thread cap

• HPX thread scheduler modified to idle/activate threads per cap

• Test example:
  – 343 domains, nx = 48, 100 iterations
  – 16 nodes of Edison, Cray XC30
  – Baseline vs. Throttled (200W per node high power cap)

No power cap
No thread cap
768 threads
Avg. 247 Watts/node
~360 kiloJoules total
~91 seconds total
~74 seconds HPX tasks
LULESH Throttled by Power Cap

200W power cap
768 threads
throttled to 220
Avg. 186 Watts/node
~280 kiloJoules total
~94 seconds total
~77 seconds HPX tasks
LULESH Performance Explanation

- 768 vs. 220 threads (after throttling)
- 360 kJ vs. 280 kJ total energy consumed (~22% decrease)
- 75.24 vs. 77.96 seconds in HPX (~3.6% increase)
- Big reduction in yielded action stalls (in thread scheduler)– Less contention for network access
- Hypothesis: LULESH implementation is showing signs of being network bound - spatial locality of subdomains is not maintained during decomposition

<table>
<thead>
<tr>
<th>Metric</th>
<th>Baseline</th>
<th>Throttled</th>
<th>% Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>1.11341E+13</td>
<td>3.88187E+12</td>
<td>34.865%</td>
</tr>
<tr>
<td>Instructions</td>
<td>7.33378E+12</td>
<td>5.37177E+12</td>
<td>73.247%</td>
</tr>
<tr>
<td>L2 Cache Misses</td>
<td>8422448397</td>
<td>3894908172</td>
<td>46.244%</td>
</tr>
<tr>
<td>IPC</td>
<td>0.658677</td>
<td>1.38381</td>
<td>210.089%</td>
</tr>
<tr>
<td>INS/L2CM</td>
<td>870.742</td>
<td>1379.18</td>
<td>158.391%</td>
</tr>
</tbody>
</table>
LULESH latest results

Figures: unmodified and power-capped (220W) executions of LULESH (HPX-5), with equal execution times. (8000 subdomains, $64^3$ elements per subdomain on 8016 cores of Edison, 343 nodes, 24 cores per node). 12.3% energy savings with no performance change.
Future Work, Discussion

- Conduct more robust experiments and at larger scales on different platforms
- More, better policy rules
  - Runtime and operating system
  - Application and device-specific (*in progress)
  - Global policies (*in progress)
- Multi-objective optimization (ActiveHarmony)
- Integration into HPX-5, HPX-3 code releases
- API refinements for general purpose usage
- Global data exchange – who does it, and when?
- “MPMD” processing – how to (whether to?) sandbox APEX
- Other runtimes: OpenMP/OMPT, OmpSS, Legion, OCR…
- Source code: [https://github.com/khuck/xpress-apex](https://github.com/khuck/xpress-apex)
Acknowledgements

• Support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics) under award numbers DE-SC0008638, DE-SC0008704, DE-FG02-11ER26050 and DE-SC0006925.

• Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. DOE’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

• This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.