Logistics

- RAMEN
  - MOU between CIS and UO IT in place
  - OS install and configuration to happen soon
- OACISS resources
  - Accounts on cerberus, centaur, minotaur
- Use OACISS resources for the GPU training
Outline

- Runtime systems
- Argo
- HPX
- MPC
Models and Abstractions

Programmer’s view

- Programming Model
- Computational Model
- Execution Model
- Architecture Model
- Machine Model

Abstractions

- Logical operation
- Translation
- System operation / interaction

- Software operation on platform
- Hardware components
- Languages Semantics

Translation

CIS 631: Advanced Parallel Computing, University of Oregon
Parallel Languages and Runtime Systems

- Parallel languages system provide users with programming abstractions
  - Language syntax and semantics
  - Embodied operational (computational) model

- If the execution environment does not support computational model explicitly, a runtime system provides a layer of necessary functionality
  - More flexible support
  - Implements functional semantics required by language
  - Implements operations needed by language features

- Parallel language systems can be mapped on different parallel runtime systems
OpenMP

- Parallel language system based on:
  - Shared memory parallelism
  - Multi-threading and multi-tasking semantics

- Runtime system provides mapping to:
  - Actual threads of execution and memory management
  - Parallel work sharing and synchronization

- Every OpenMP compiler has a runtime system
  - OpenUH
  - GNU gcc (GOMP)
  - Intel Icc
  - PGI
  - …

- ECP SOLLVE project
  - Scale OpenMP with LLVM

Challenging to build tools!
MPI

- MPI can be considered to have a runtime
  - Implements the MPI library
  - Implements startup and termination
  - Manages buffers, communicators, ...
  - Implements collective operations, ...
  - Works with multiple languages and threads/processes

- Provides alternative support different hardware
  - Networks
  - Shared memory optimization

- Exascale MPI ECP project looks at large scale
  [https://www.exascaleproject.org/project/exascale-mpi/](https://www.exascaleproject.org/project/exascale-mpi/)
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)

http://xstack.sandia.gov/xpress/
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 to write fully asynchronous code using hundreds of millions of threads
  - Unified syntax/semantics for local/remote operations
  - HPX-3 (LSU, C++ language/Boost)
  - HPX-5 (IU, C language) (defunct)
OpenX Stack

- Integrated exascale software stack for ParalleX
- Higher-level languages support
- HPX runtime implementation
- Autonomic Performance Environment for Exascale (APEX)

![Diagram of OpenX Stack and its components]

- Legacy Applications
  - OpenMP
  - MPI
- New Model Applications
  - Domain Specific Language
  - Domain Specific Active Library
  - Compiler
  - XPI
- Metaprogramming Framework
- Domain Specific Language
- Metaprogramming Framework
- AGAS name space processor
- LCO dataflow, futures synchronization
- Lightweight Threads context manager
- Parcels message driven computation
- PRIME MEDIUM Interface / Control
- Distributed Framework
- Operating System
  - Task recognition
  - Address space control
  - Memory bank control
  - OS thread
  - Network drivers
  +10^6 nodes \times 10^3 cores / node + integration network

CIS 631: Advanced Parallel Computing, University of Oregon
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
Dynamic Execution Strategy

(a) local thread instantiation
(b) remote thread instantiation
(c) remote atomic memory operation
(d) depleted thread activation
(e) dataflow object trigger
(f) future value access
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 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.

* From XPRESS proposal
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) tracks system resource assignment, utilization, job contention, overhead
  - Runtime (HPX) tracks threads, queues, concurrency, remote operations, parcels, memory management
  - ParalleX, DSLs and legacy codes allow language-level performance semantics to be measured
APEX Introspection

- APEX collects data through “inspectors”
  - **Synchronous** uses an event API and event “listeners”
    - Initialize, terminate, new thread
    - Timer start, stop
    - Sampled value
    - 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 (e.g., /proc/stat)
APEX Event Listeners

- **Profiling listener**
  - Start event: take timestamp, return profiler handle
  - Stop event: take timestamp, put profiler object in a queue for back-end processing, return
  - Sample event: put the sample in the queue
  - Consumer thread: process profiler objects and samples to build statistical profile

- **Concurrency listener**
  - Start event: push timer ID on stack
  - Stop event: pop timer ID off stack
  - Consumer thread: periodically log current timer for each thread, output report at termination
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

![Graph showing the relationship between number of worker threads and runtime for 1D stencil code.](image)
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
1d_stencil Baseline

- 48 worker threads (w/ hyperthreading)
- Actual concurrency much lower
  - Many worker threads are spinning waiting for work to become available
- Large variation in concurrency over time
  - Tasks waiting on prior tasks to complete

Where calculation takes place

Event-generated metrics

138 secs

100,000,000 elements, 1000 partitions

- Many worker threads are spinning waiting for work to become available
- Large variation in concurrency over time
1d_stencil Optimal # of Threads

- 12 worker threads
- Greater proportion of threads with work
  - Less interference between active threads and threads waiting for work
  - Edison has 24 physical cores; idle hardware threads >24 are hyperthreads which may interfere with active threads
- Much faster
  - 61 sec. vs 138 sec.
**1d_stencil Optimized with APEX**

- Initially 48 worker threads
- Discrete hill climbing search to minimize average # of pending tasks
- Converges on 13 vs. 12
- Nearly as fast as optimal
  - 64 seconds vs. 61 seconds

![Graph showing concurrency, power, and time for 100,000,000 elements, 1000 partitions.](chart.png)
Argo

- A DOE exascale operating system and runtime software research project
  - Labs: ANL, LLNL, PNNL
  - Universities: UChicago, UIUC, UO, UTK

- ECP project focus on Argobots
  - Lightweight low-level threading/tasking framework
  - Separation of abstraction and mapping to implementation
  - Execution streams for progress
  - Work units execute to completion

http://www.argo-osr.org
How is Argo Different?

- Assume dynamic and variable computing performance
- Manage power as a system resource
- Support hierarchical, active control systems
- Provide global performance (introspection) and management (control & fault) backplane
- Provide extreme on-node thread/task parallelism linked to memory model
- Extend Linux to provide HPC containerization, schedulers, and hierarchical memory management
- Provide HPC interface to NVRAM
- Demonstrate with multiple parallel language systems
**Argo NodeOS**

- Single kernel image, Linux-based
- Partition hardware resources using containers
  - ease of management, potential scalability improvements
- Small-core-count ServiceOS for overall node management
- Compute containers for running application code
- HPC-specific improvements (memory, scheduler)
MPC

- See MPI Forum talk