Overview

- Future large scale systems
  - Large number of cores per node
  - Heterogeneous: Cell, GPU, Larrabee, Nehalem
  - Asymmetric: Nehalem, GPU, torus network

- Efficient utilization requires:
  - Multiple levels of parallelism
  - Hybrid execution
  - Efficient intra-node communication
  - Adaptive scheduling

- Multiple projects:
  - OS and parallel runtime interface development and co-design: load balancing, cooperative scheduling, performance introspection
  - Efficient intra-node communication for PGAS programming languages: collective operations, mapping of language threads to the OS-level execution contexts

Process – Shared Memory Overview

- Project Goals:
  - Improve performance of UPC on multi-core shared memory machines
  - Improve interoperability (Hybrid Execution: MPI, OMP, UPC)

- Our work:
  - Investigate mapping of the Language threads to the OS threads
  - We implemented Process - Shared Memory (PSHM) execution in UPC (available in the current release)

The PGAS execution model is amenable to two-level scheduling

Pthreads-Process Runtime Comparison

- Previous BUPC releases used pthreads for shared memory communication within a single SMP node
- Many libraries are not thread safe (FFTW, C I/O functions on certain OSs) – interoperability problems
- Pthreads share the entire address space, while processes share only certain memory regions
- Pthreads – PSHM behavior differences
  - Pthreads share network connections (e.g. InfiniBand) when hardware allows 1 connection per process
  - Different context-switch overhead for thread/process
  - Any thread can serve Active Messages on behalf of any other thread, since they share the address space

Pthreads-Process Performance

Microbenchmarks and NPB experiments are conducted on 2-node Intel Tigerton (4 socket, 4 core) cluster. Machines are directly connected via InfiniBand.

The PGAS execution model is amenable to two-level scheduling

Auto-tuned Multicore Collectives

- Open questions:
  - Collective interface design for global address space
  - Collective interface design for one-sided communication

- Strict Synchronization
  - Data movement can start only after all the threads arrive at the collective and must be done before the first thread leaves the collective
  - Usually implies a barrier before and after the collective
  - Easy to understand but often over synchronizes the operation

- Loose Synchronization
  - Data movement can start after any thread enters the collective and can continue until last thread leaves
  - Allows user to aggregate synchronization costs across many operations
  - Enables better use of memory system
  - More difficult to program
  - “Traditional pthread barriers” yield poor performance
  - Performance penalty for picking bad algorithm is large
  - Loose synchronization yields performance improvements, enables pipelining