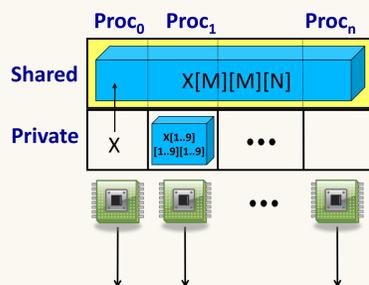


Introduction

This poster describes our work on Scioto, a new parallel programming model that provides scalable support for task parallel programming on distributed memory clusters. Scioto's task model complements existing Partitioned Global Address Space (PGAS) data models to form a complete environment for expressing and managing irregular and dynamic parallelism. The Scioto programming model is supported by a scalable runtime system that provides dynamic load balancing and improves communication overheads by co-locating tasks with data on which they operate. We present an evaluation of Scioto on several benchmarks including the MADNESS computational chemistry kernel and demonstrate strong scaling and high efficiency on an 8,192 core cluster.

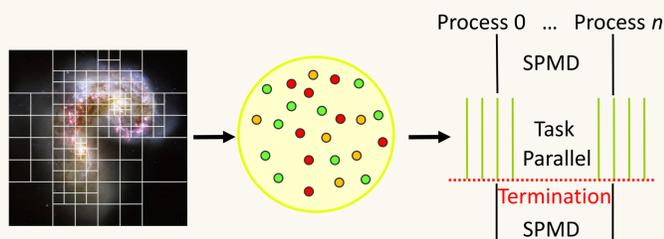
PGAS Models and The Asynchronous Gap

- PGAS models provide an asynchronous irregular data model
 - E.g. Global Arrays, UPC, CAF
- Computation model is still regular, process-centric SPMD
 - Irregularity in the data can lead to load imbalance



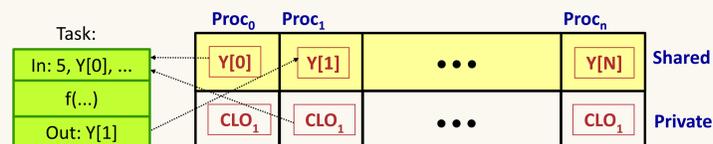
- Scioto extends PGAS models to bridge asynchronous gap
 - Dynamic task-based view of the computation

Scioto: Scalable Collections of Task Objects



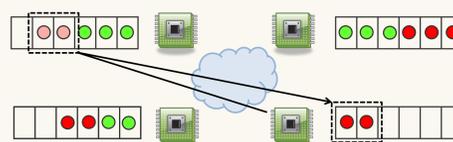
- Programmer expresses the computation as collection of tasks
 - Tasks operate on data stored in PGAS (Global Arrays)
 - Executed in collective task parallel phases
- Runtime system manages task execution / task parallel phases
 - Load balancing, locality optimizations, fault resilience, etc

Scioto Task Model



- Task Inputs: Global data, Immediates, Common Local Objects (CLO)
- Task Outputs: Global data, CLOs, Child tasks

Runtime System Design



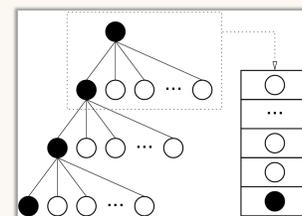
- Per-process ARMCI circular task queues for efficient one-sided access
 - Queues are prioritized by affinity
 - Use the work first principle (LIFO task execution)
 - Load balancing off the tail via random work stealing (FIFO stealing)

Scalable Work Stealing

- Enhancements to enable efficient scaling to 8,192 cores
 - Highest known scaling for work stealing
- 1. Split work queues
 - Optimize local accesses, reduce locking on critical path
- 2. Work splitting: Steal-half
 - Reduce search time, improve work distribution
- 3. Aborting lock operations
 - Abort long waits on exhausted resources

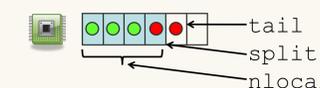
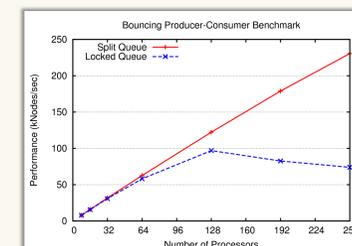
Experimental Setup and Benchmarks

- HP Infiniband Cluster
 - 2,310 Nodes, 2x2.2GHz 4-core AMD
- BPC: Bouncing Producer Consumer
 - Producer task migrates due to load balancing operations
- MADNESS: Comp. chemistry kernel
 - Project 3-d function into oct-tree spatial representation
- UTS: Unbalanced Tree Search Benchmark
 - Exhaustive parallel DFS on highly unbalanced tree



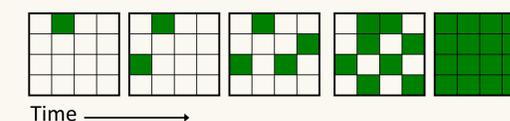
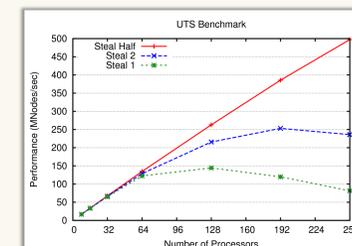
1. Optimize Local Accesses: Split Queues

- Queues are split into two parts:
 - Private: Local-only
 - Shared: Any, locked
- Removes locking from critical path
 - Local enqueue/dequeue
- Periodically move split as computation progresses
 - Reacquire work
 - Release work (lockless)



2. Reduce Search Time: Work Splitting

- Problem:** Search time grows with system size
- Strategy:** Divide tasks evenly between victim and thief
 - Double number of work sources after each step
 - Reduce avg. time to find work to $\log(ncpus)$



3. Manage Contention: Aborting Steals

- ARMCI Locks: Bakery Algorithm
 - Take a ticket, wait in line
 - Fair, but if victim runs out of work must still wait to give up ticket
- Spinlocks:
 - `while(!atomic_swap(lock))`
 - Can give up at any time
- Spinlocks + Aborting Steals:
 - Periodically check if we should abort lock()
 - Avoid waits on stale resource

