ORACLE®

NUMA-aware Exchanger

Dave Dice
» Oracle Labs - Scalable Synchronization Research Group
» http://blogs.oracle.com/dave

Exchanger - Interface

- java.util.concurrent.Exchanger
- message = ex.exchange(message)
- fully synchronous rendezvous
- symmetric : exchange instead of put/take
- useful for buffer exchange and to avoid memory bloat issues associated with in-flight data in queues

Existing Implementation

- Array of slots
  - resize automatically : 1 to 32
- hashIndex() helper function
  - identifies preferred slot for thread T
  - based on stable hash of Thread.getId()

Existing Implementation

- exchange()
  - start at caller's preferred slot
  - if pending offer in that slot then remove and complete rendezvous
  - install offer into array
  - linger for a short time at that index
  - if no rendezvous, rescind offer
  - move to lower index, retry - scan
  - spin-then-park at slot [0]
  - converge toward 0 for progress

Index collisions

- Too few : increased scanning
- Too many : impede progress - chokepoint
- Ideally want just the right level of collisions to accomplish rendezvous
- Prefer uniform arrival rate over all indices

NUMA awareness

- Exchanger is NUMA/<u>NUCA</u>-oblivious
- Ideally
  - prefer intra-node exchange : endogamous
  - scanning should minimize nodes visited
- Reduce coherence traffic
  - Write sharing & invalidation
  - Interconnect traffic is limiting factor for large NUMA systems
  - Interconnect capacity not increasing at same rate as cores
  - Bandwidth and latency concerns

T5440

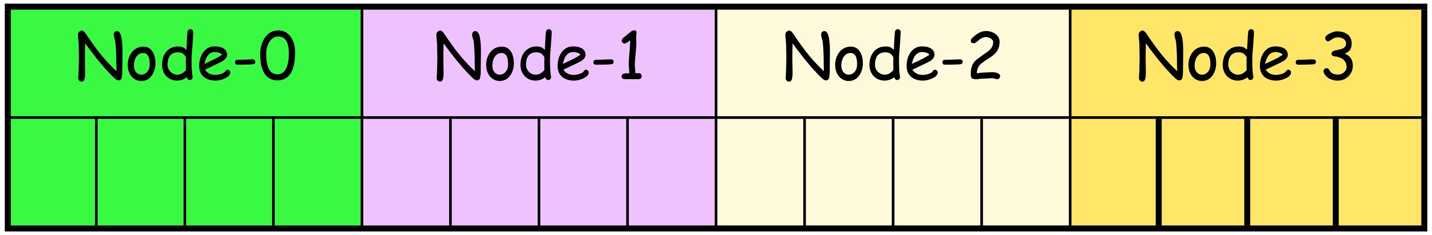
- 4-socket Niagara T2+ : 256x
- Each socket is a NUMA node
- Zambezi coherence hub
- 4 independent coherence planes
- Unloaded round-trip memory communication times :
  - same core : 130ns
  - same socket : 165ns
  - inter-socket : 650ns - NUCA

CPUIDs

- Geographic CPUID layout on T5440
  - [0,256)
  - Node:2 | Core:3 | Pipeline:1 | Strand:2
- Solaris scheduler disperses threads over CPUs to maximize “distance”
  - distal placement
- Access CPUID in user-land via per-thread schedctl structures
- JNI DirectBuffer access to schedctl
  - JIT : 2 LDs to query CPUID
  - efficient cpuid() primitive
  - never leave managed code

NUMA friendly Exchanger

- Partition array into bands - stripes
  Geographically partitioned array
- Form communication cliques
- Simple linear mapping from cpuid() to index
- Index := (cpuid() * ArraySize)/maxcpuid
  Scale CPUID into array index
- Trivial modification : 6 lines - hashIndex()

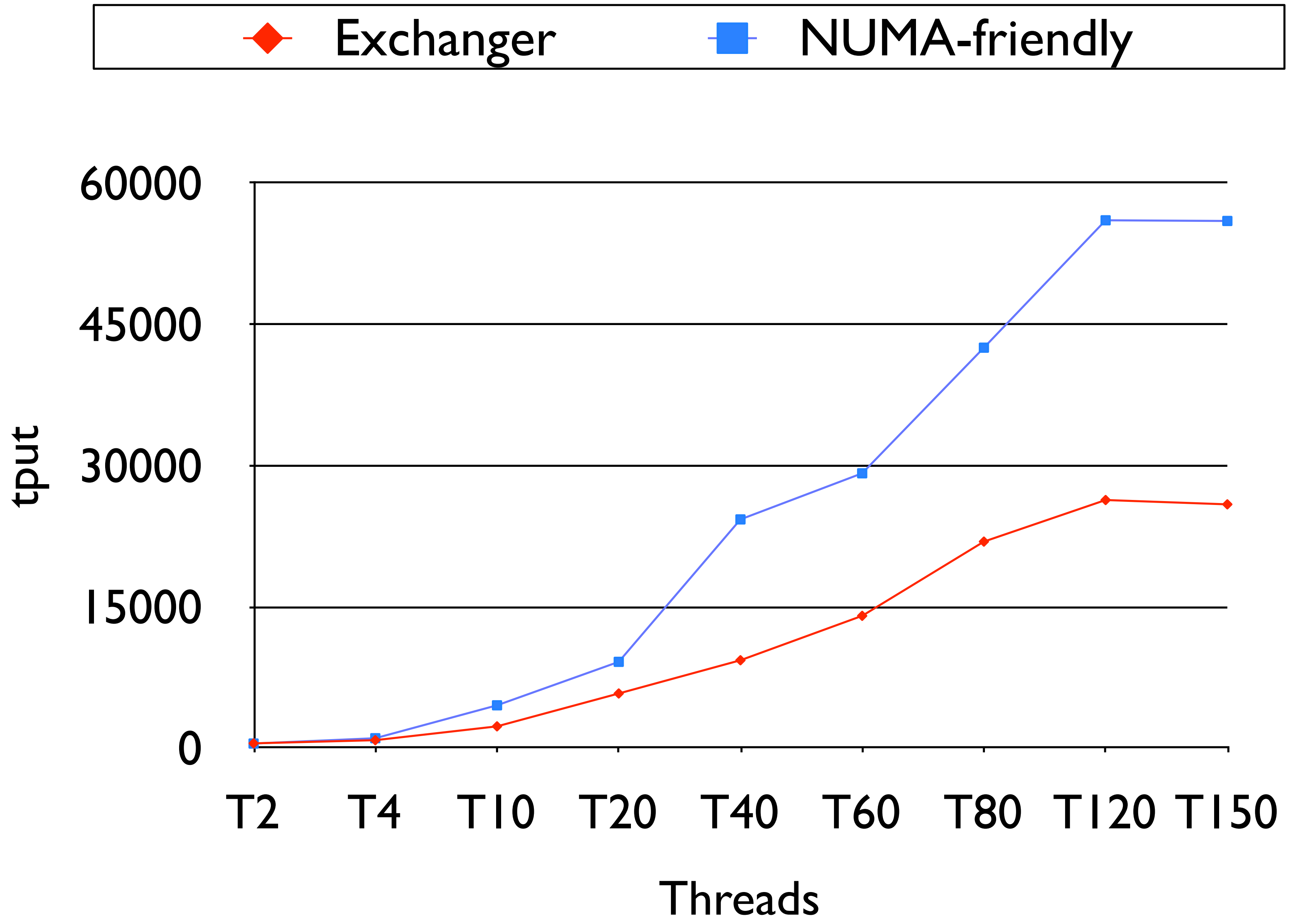


Outcome ….

- A given array index is mapped to by threads that reside on the same node : encourages local exchange
- Adjacent indices are more likely associated with the same node : scanning more efficient - tend to touch local indices 1st

Benchmark

- Spawn T concurrent threads
- All loop calling exchange() on single global exchanger instance
- 10 second measurement intervals
- report aggregate throughput in messages exchanged per msec
- 10 sub-trials for warm-up
- report median of last 3
- Threads on X-axis, throughput on Y
- +UseCondCardMark


Exchanger
NUMA-friendly
60000
45000
30000
15000
0
tput
T2
T4
T10
T20
T40
T60
T80
T120
T150
Threads

What's next

- Fork-join : encourage node-local stealing
- CPU-specific caches or counters
  - Often better than thread-specific
- Concurrent data structures striped by CPUID

Low-level Interface

- intrisified Unsafe.getCpuId()
  - schedctl on Solaris
  - RDTSCP on other x86
- Topology discovery
  - CpuIdToNodeId(cpuid)
  - AvailableNodes()
- Distance metric : distanceTo(A,B)

Alternative to distanceTo()

- Proximity map : Map[cpuid]
- constructed by JVM
- Converts actual cpuid into abstract cpuid
- Numerically close abstract values tend to be nearby in system topology
- Usually but not always : #63, #64
- Identity map for Solaris/SPARC
- Potentially easier to use than distanceTo()
- Abstract ID space is a continuum
- Easy to map abstract IDs onto concurrent data structure to provide coarse striping

-XX:+UseCondCardMark

- Reduces false sharing in GC card table
- Most card table stores are redundant
- Don’t store if already marked dirty
- 32KB “card page”

What about binding?

- Actively control thread placement
- Affinity APIs vary by platform
  - advisory vs mandatory
- Need to understand CPUID mappings and platform topology
- Inimical to dynamic reconfiguration and virtualization
- Tension : virtualization introduces new layer while, for performance, we'd like JVM to be even more intimate with system

What about binding?

- OS scheduler or hypervisor has a better view of system than does a process locally
  - better able to manage thread placement
- In practice multiple mutually unaware JVM processes that use binding will cause a mess.
  - some CPUs over-saturated
  - others under-utilized
  - begs for global resource coordination
  - don't try to make global resource decisions with local information

Thank you

- http://blogs.oracle.com/dave
- http://blogs.oracle.com/dave/entry/solaris_scheduling_and_cpuids
- http://blogs.oracle.com/dave/entry/false_sharing_induced_by_card

Backup

- balls-and-bins probabilistic model
- busstat and cpustat confirm reduction in coherence traffic
- NUCA instead of NUMA
  - Our technique leverages ambient memory placement : passive
  - Agnostic regarding home nodes
  - concerned with communications costs : coherence misses

Aside

- Interesting JUC exchanger hashIndex() issue
- Thread creation order correlated with getId values and thus hashIndex(getId) value
- Thread creation order also correlated with relative NUMA placement on large systems like 256x T5440
- hashIndex(): input values that are multiples of 32 tend to map to same index when "max" is set to 31
- hashIndex(): input values that are multiples to 4 tend to map to same quadrant in slot array

Aside

- Inadvertent correlation
  - Thread launch order
  - Thread getID value
  - hashIndex(getId)
  - NUMA placement of thread
  - index into JUC exchanger slot array
- Results in unexpected partitioning of slot indices - unexpected geographic relationship