

Lecture Topics

- performance reasoning for shared memory threads (Lab 3) (wrap-up)
- scheduling scope implications in Posix threads

Administrivia

- lab 3: warnings fixed on Ipaint machines; grab a new copy if desired

- extra complexity in shared memory systems
 - false sharing
 - hardware defines size of shared data
 - e.g., cache line, page
 - exact size is not well-defined
 - hardware protocol thrashes for finer-grained sharing
 - e.g., two processors write disjoint words in a cache line
 - line ping-pongs between caches
 - no actual sharing involved
 - but a huge performance penalty
 - performance tuning: sharing issues are not local in code
 - how do you reason about data sharing issues in parallel code?
 - sharing depends on data layout and parallel operations
 - parallel operations are regions of code
 - data layout is defined by data structures
 - not easy to optimize one operation independently of others
 - not easy to optimize entire body of code simultaneously
 - scalability concern: hardware coherence is lazy message-passing
 - for any cache (shared memory or otherwise)
 - data are requested when the load instruction issues
 - takes time to request and receive into cache
 - prefetching can help in some cases
 - becomes a problem when time to memory varies widely
 - often more efficient to push data in advance
 - avoid added latency

- resource contention
 - resource use correlated by bulk-synchronous structure
 - example
 - two data layouts to avoid false sharing
 - phases
 - » execute first operation
 - » transform data
 - » execute second operation
 - use barriers to prevent overlap
 - performance analysis
 - operations are compute-intensive
 - data transform is memory-intensive
 - overlapping phases allows more efficient use of both
 - may want to weaken synchronization
 - once first operation complete for part of output
 - that output can be transformed to new layout
 - once input to second operation is complete
 - second operation can be launched
 - easier to enable with asynchronous access to data
 - allowing first/second operations to overlap
 - not clearly useful (both compute-intensive)
 - may be detrimental (delay in first operation leads to completion delay)

- subtle problem: self-synchronization of fine-grained resources
 - Brewer & Kuszmaul example
 - how it works in general
 - solution? let me know if you find a good one

- review summary of challenges in context of shared memory threads
 - atomicity & precedence/dependence
 - shared address space gives free “broadcast”
 - no need to identify recipient(s)/competition
 - relatively easy to express/enforce
 - some examples
 - locks for atomicity
 - condition variables for dependence
 - complicated by asynchronous access model
 - easy to overlook
 - often not easy to detect abuse
 - inheritance anomaly
 - original formulation, so more or less unchanged in difficulty
 - avoids extra complexity induced by message-passing (for example)
 - but don’t forget to solve false sharing, too
 - algorithm vs. system
 - sharing
 - scheduling
 - synchronization
 - make them all work well at the same time
 - determinism
 - good luck! the hardware’s not going to help...
 - (ok, there has been some recent research on replay)

- Scheduling scope and debugging value
 - Process vs system scope
 - Implications for simple example
 - Use as a debug tool
-
- Note: still need to test between turning over to system scope and releasing...