Lecture Topics

- performance reasoning for shared memory threads (wrap-up)
- detecting errors in synchronization
- Eraser + implementation

Administrivia

- [none]

- extra complexity in shared memory systems (cont's)
  - 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-intense)
    - 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)

Detecting Errors in Synchronization

- the basic problem
  - once variables may be visible to other code/unsynchronized
  - need to worry about the following scenario
    - two blocks of code, B and C
    - each block reads some variables: R_B and R_C are sets
    - each block writess some variables: W_B and W_C are also sets
  - conflict/data race may happen
    - if one block writes a variable read/written by the other
    - that is
      - $W_B \cap (R_C \cup W_C) \neq \varnothing$ OR
      - $W_C \cap (R_B \cup W_B) \neq \varnothing$

- monitors (Hoare, 1974)
  - a group of shared variables and procedures
  - protected by a lock (called a monitor)
  - variables are scoped within the procedures
  - lock serializes execution of all procedures
- problem with monitors
  - difficult to compose abstractions
  - need abstraction ordering to avoid deadlock
- still used, but not usually strictly
  - synchronized keyword in Java
  - not required/automatic
  - can be per-object or per-class