Outline

- Announcements
  - no lecture on 04/21 (work on project)
  - project presentations on 04/28

- Last 5 lectures: Middleware for clusters
  - communication: messaging, software shared memory
  - fault-tolerance and availability

- This lecture: Resource coordination
  - problem: each node in the cluster runs a full independent OS
  - background: kernel- and application-level scheduling problems
  - case studies
    - work-stealing schedulers (application-level) [Harris]
    - implicit scheduling (kernel-level) [Holger]
    - dynamic coscheduling (kernel-level) [Holger]
Resource Coordination in Clusters: Challenges

- Independent OS on each node
  - processes of a parallel job are not co-scheduled
  - interference from other jobs (multiprogramming), daemon activities

- Interactivity
  - interaction between sequential and parallel workloads
  - non-availability of machine: parallel job must migrate elsewhere

- Heterogeneity
  - classical load-balancing problems

Coscheduling versus Local Scheduling

- Frequently synchronizing applications exhibit 50x slowdown
  - becomes worse with more competing processes
- Coscheduling skew impacts performance less than 5%
  - processes do not need to be precisely coscheduled
Impact of daemon activity

- Slowdown not significant for typical daemon activity
  - inter-arrival times of 10s, duration of 5ms result in <10% slowdown
- However, if daemon activity is longer, coscheduling may be necessary

Two-level Scheduling Problem

- Kernel-level scheduler
  - allocates OS scheduling quanta to user processes
- User-level scheduler
  - allocates (user-level) threads to OS scheduling quanta

- Approaches to overcome performance loss due to local scheduling:
  - influence the kernel-level scheduler
    (implicit scheduling, dynamic coscheduling)
  - influence the user-level scheduler
    (work stealing)
Work-Stealing Schedulers

- Programming model
  - dynamic pool of user-level threads
    - number, granularity independent of number of processors, processes
  - processes act as workers which pick a thread from the pool and execute it
  - “work stealing” because
    - management of the thread pool is distributed among the processes
    - picking a thread from the pool: “stealing” a thread from another process

Performance of Work-Stealing Schedulers

- Application decomposed into P processes
  - run on a dedicated 8-processor machine

[Figure 1 from “The Performance of Work Stealing …” paper]
Work-Stealing: Implementation Challenges

- All data in the program must be accessible (even if a process is not scheduled)
  - all data is replicated in all processes (e.g., the Cilk runtime)
  - hardware shared memory (as in the paper)
  - alternately, cluster NI must provide this functionality
    - Myrinet: Programmable NI processor
    - SHRIMP, Memory Channel: Support for remote memory operations

- Access to the shared thread queue
  - processes access concurrent data structures within a critical section
  - however, the kernel can preempt processes which hold the lock
  - 3 choices
    - spinning locks (test-and-set operation against lock state)
    - blocking locks (kernel manages mutex operations)
    - lock-free (using compare-and-swap, LL/SC, etc.)

Lock-free Shared Queue

[ Figures 1, 2 from the “Thread Scheduling for Multiprogrammed Multiprocessors” paper ]
Deque from a Shared Queue

[Figure 2 from “The Performance of Work Stealing …” paper]

(a) Spinning lock.

(b) Blocking lock.

(c) Lock-free.

Improved Lock-free Design

- **Problem:** Scheduled processes spin waiting for non-scheduled processes to put work into queue

- **Insight:** Current quantum allocation is better used by the process which is being waited upon
  - spinning process yields its scheduling quanta (using **yield**)
  - to facilitate scheduling of processes with work, idle processes reduce own priority (using **priocntl**)

- **Consequence:** Processes which have work receive quantum allocations from those that do not
  - ideally would like to direct OS to give own quanta to a specific process
Theoretical Results

- Computation
  - work: $T_1$, critical-path length: $T_\infty$
  - number of processes: $P$

- Environment
  - $P_A$: average number of processors on which the computation executes
    - average of $p_i$ (= number of processes scheduled in step $i$)
  - kernel is an adversary
    - controls how many and which processes are scheduled in a step

- Lock-free implementation of the work stealing scheduler
  - $E(T) = O(T_1/P_A + T_\infty P/P_A)$ assuming that the kernel is a
    - benign adversary and the yield system call does nothing
    - oblivious adversary and there exists a yieldTo system call
    - adaptive adversary and there exists a yieldToAll system call

Lecture Summary

- Resource coordination in clusters
  - ✔ independent OS per node
  - interactive job requirements
  - heterogeneity

- Overall problem can be viewed as a two-level scheduling problem
  - kernel-level: implicit scheduling, dynamic coscheduling
  - user-level: work stealing
  - both achieve multiprogrammed performance which is close to that of
dedicated environments

- Research issues
  - can similar techniques be used to coordinate other node resources
    (memory, disk, network interface channels, etc.)?
  - would a better overall solution combine kernel- and user-level schedulers
    (e.g., scheduler activations, etc.)?
Next Lecture

- Putting it all together: Cluster case studies
  - *Parallel Computing on the Berkeley NOW*, Culler et al.
  - *The Inktomi Experience*, Brewer

- Conclusions