Lecture 12: Multiprocessor Scheduling II

CSC 469H1F / CSC 2208H1F

Fall 2007

Angela Demke Brown

Parallel Job Scheduling

- Recall threads in a parallel job are not independent
  - Scheduling them as if they were leads to performance problems
  - Want scheduler to be aware of dependences
- Forms of scheduler-awareness
  - Know threads are related, schedule all at same time
  - Know when threads hold spinlocks and don't deschedule lock holder
  - Know about general dependences

Using thread relations

- Space sharing (typically supercomputers)
  - At job creation, specify number of threads
  - Scheduler finds set of CPUs
    - May negotiate with application
      - "I can't get you 512 CPUs right now, would you like to wait or run with only 8?"
      - Many parallel applications can choose the # of threads
  - How should scheduler choose jobs to assign to CPUs? What is optimal (in terms of average wait time)?
    - Uniprocessor scheduling → shortest job first (shortest expected next CPU burst)
    - MP version → smallest expected number of CPU cycles (cycles == num_cpus * runtime)
      - FCFS with backfilling is hard to beat
      - Backfilling requires estimates of run time, which could be used to implement MP version of SJF

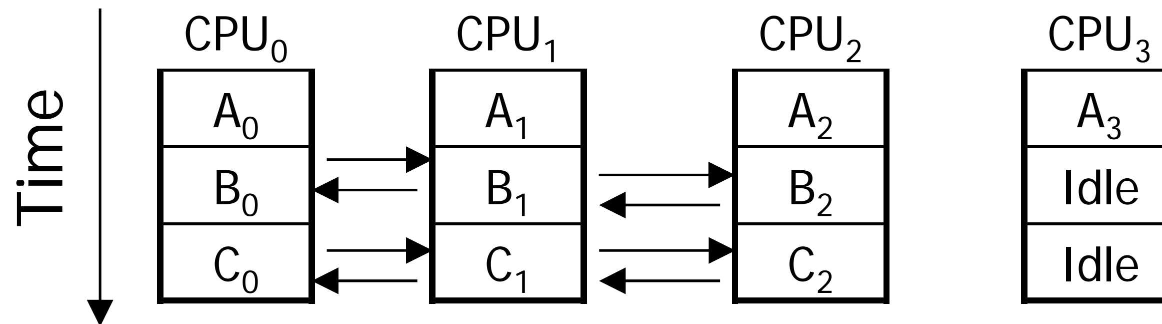
Estimating Runtime

- Estimates typically come from users who submit the jobs
  - Low estimates make it easier to do backfilling
  - But cause trouble with reservations if not accurate!
  - Soln: kill jobs that exceed estimate
- How accurate are user estimates?
- Can automatic estimates based on history do better?
- How much does it matter?

Parallel Time Sharing

- Each CPU may run threads from multiple jobs
  - But with awareness of jobs
- Co-scheduling (Ousterhout, 1982)
  - Identify "working set" of processes (analogous to working set of memory pages) that need to run together
- Gang scheduling
  - All-or-nothing → co-scheduled working set is all threads in the job
  - Get scheduling benefits of dedicated machine
  - Allows all jobs to get service
- 2-D Bin packing problem to fill available CPU slots with runnable jobs

Gang Scheduling Example



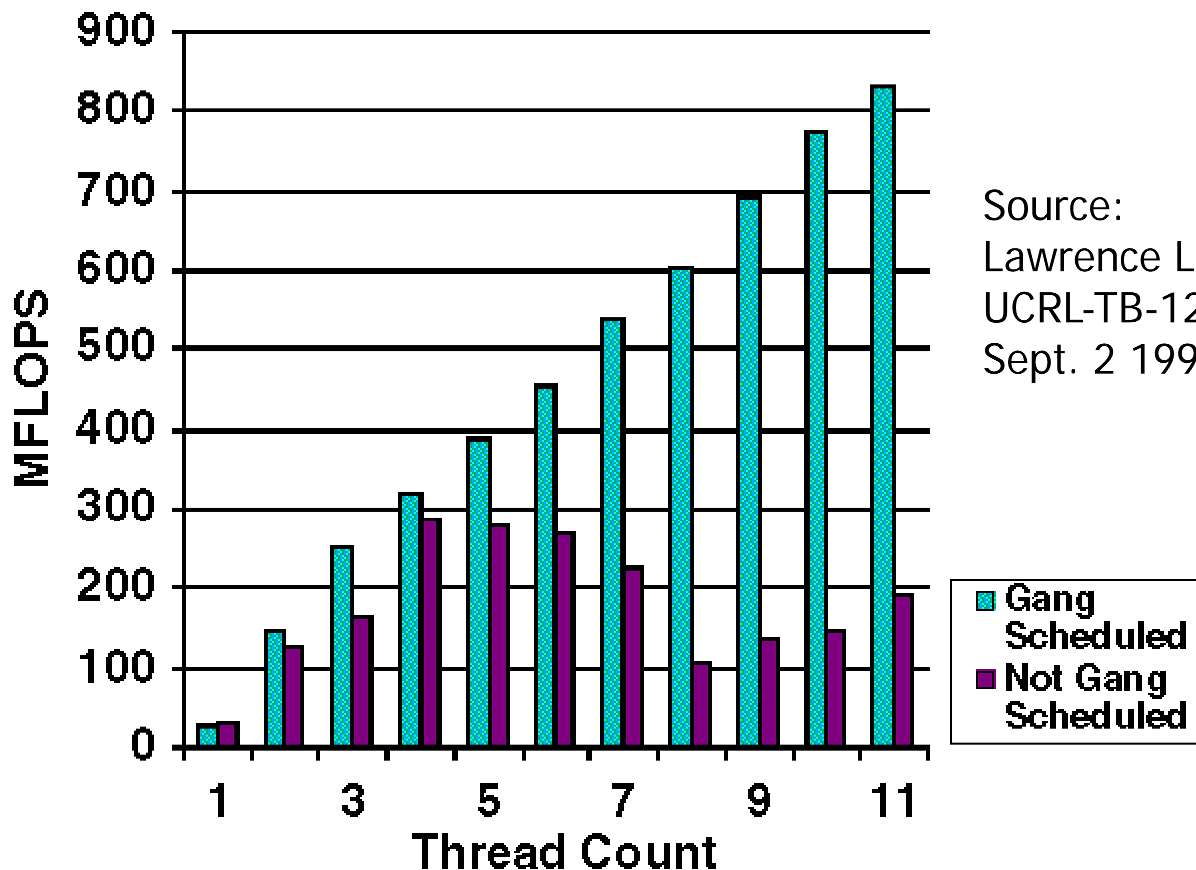
- Multiprogramming level is typically controlled by either:
  - Monitoring memory demand, or
  - Fixed number of slots (rows)
    - E.g. IBM LoadLeveler Gang Scheduling allows up to 8 sets of jobs to be multiprogrammed on a set of CPUs

Gang Scheduling Issues

- All CPUs must context switch together
  - To avoid fragmentation, construct groups of jobs that fill a slot on each CPU
    - E.g., 8-CPU system, group one 4-thread job with two 2-thread jobs
  - Inflexible
    - If 4-thread job blocks, should we block entire group, or schedule group and leave 4 CPUs idle?
- Alternative 1: Paired gang scheduling
  - Identify groupings with complementary characteristics and pair them. When one blocks, the other runs
- Alternative 2: Only use gang scheduling for thread groups that benefit
  - Fill holes in schedule with any single runnable thread from those remaining

Example: Effect of Gang Scheduling

- LLNL gang scheduler on 12-CPU Digital Alpha 8400
  - Parallel gaussian elimination program
  - http://www.llnl.gov/asci/pse_trilab/sc98.summary.html



Source:
Lawrence Livermore Natl Lab
UCRL-TB-122379-Rev2
Sept. 2 1998

Knowing about Spinlocks

- thread acquiring spinlock sets kernel-visible flag
- Clears flag on release
- Scheduler will not immediately deschedule a thread with the flag set
  - Gives thread a chance to complete critical section and release lock
  - Spinlock-protected critical sections are (supposed to be) short
  - Does not defer scheduling indefinitely

Knowing General Dependences

- Implicit Co-scheduling (Arpaci-Dusseau et al.)
- Designed for workstation cluster environment
  - Explicit messages for all communication/synchronization
  - MUCH more expensive if remote process is not running when local process needs to synchronize
- Communicating processes decide when it is beneficial to run
  - Infer remote state by observing local events
    - Message round-trip time
    - Message arrival
- Local scheduler uses communication info in calculating priority

OS Noise

- Or: how to schedule OS activities
- Massively parallel systems are typically split into I/O nodes, management nodes, and compute nodes
  - Compute nodes are where the real work gets done
  - Run customized, lightweight kernel on compute nodes
  - Run full-blown OS on I/O nodes and mgmt nodes
  - Why?
- Asynchronous OS activities perturb nice scheduling properties of running jobs together
  - Up to a factor of 2 performance loss in real large-scale jobs
  - Need to either eliminate OS interference, or find ways to coordinate it as well