CMSC 714
Lecture 17
Short Term Scheduling

Alan Sussman

Notes

- Midterm exam on Nov. 17
  - sample exam questions posted
- GPU cluster accounts ready
  - for those who asked, account should use your class account ID and password (from bug cluster)
  - log into chimerasub{00,01}.umiacs.umd.edu cluster nodes are chimera00 – chimera31, accessed via batch scheduler (same as for bug cluster)
  - more chimera cluster info at https://wiki.umiacs.umd.edu/umiacs/index.php/ChimeraCluster

Scheduling of cluster resources

- Setting is time sharing of compute resources to run parallel programs, on clusters or dedicated parallel machines
- To improve utilization, and do better in both throughput and response time than *gang scheduling*
  - also called co-scheduling, and means that all the processes for the parallel job are scheduled (context-switched) at the same time
  - also want to run serial jobs on the same machines
- Want to target common SPMD parallel program bulk synchronous execution model
  - alternating phases of computation and communication (maybe I/O too), with maybe a barrier before and/or after each communication phase
- Both perform extensive simulation studies, varying several parameters and using different scheduling algorithms
  - in the end both find that it's possible to do better than co-scheduling, but each ends up with a somewhat different best algorithm – starting from different assumptions, and different environments (papers are 5 years apart)

Distributed Scheduling paper

- Want to use local schedulers on each machine/OS
  - a priority-based process scheduler, to improve throughput (vs. round-robin) – goal is fair and efficient scheduling
    - jobs that block frequently (e.g., because of I/O) get high priority when they are unblocked
    - jobs that do a lot of computation get lower priority
    - but that is not enough to coordinate the processes of a parallel program that communicate or synchronize
- The basic new idea is to determine the amount of time a process that communicates/synchronizes with other processes should spin-wait before blocking
  - upon blocking, the local scheduler context-switches to another process
  - the goal is that if the processes are already coordinated in time, the spin wait will be long enough for the communication to happen w/o blocking, and if not the blocking will end when the communication operation can complete, helping to coordinate the processes for future communication operations
  - called *implicit scheduling*

Distributed Scheduling

- Several scheduling variations tested, for 3 communication patterns (barrier, NEWS, transpose)
  - Local scheduling with immediate blocking
    - better than co-scheduling for coarse and medium-grain computations with high load imbalance among processes
    - always worse for fine-grained computations, for all communication patterns
    - rest of experiments concentrate on low latency networks with high context-switch costs, since that's where immediate blocking does worst (for fine-grained comps)
  - Static blocking - two-phase fixed-spin then block
    - need to go to long spin times to get coordinated better, but still much worse than co-scheduling for fine-grain, and with load imbalance,
  - Adaptive blocking –always spin for at least 2 times context-switch time (skew), block if detect too much load imbalance
    - local algorithm approximates load imbalance by locally sampling time for barriers to complete
    - global algorithm samples barrier time at root of barrier, sends the info with the barrier completion message
- At worst, can always get performance within 35% of co-scheduling, for programs that do a lot of communication/synchronization

Workload/System Parameter Study

- Comprehensive study of dynamic scheduling mechanisms against static co-scheduling, via simulation
  - vary job arrival rate, job size (# CPUs and execution time), job resource requirements (CPU vs. communication vs. I/O), job mix, work imbalance and skew between between processes in a job, system parameters (# jobs per node, # CPUs, OS context switch and interrupt cost
  - use different heuristics to improve *Periodic Boost* mechanism
  - workload based on real workload from LLNL supercomputer
  - user-level messaging, so communication operations do not have to block the process (not like I/O operations)
- Local scheduler is priority-based again, with a multi-level feedback queue at each node
  - more details than in distributed scheduler paper, but works in similar way
  - tries to balance compute vs. I/O bound jobs, since I/O bound jobs will typically end up with higher priority (but won't compute for long before blocking)
  - default user-level send op appends to queue in memory, receive spins until message arrival

Dynamic scheduling strategies

- Spin-block (SB)
  - spin for a fixed time, before blocking – question is how to choose spin time (see dist. scheduling paper)
    - here set to somewhat larger than latency of expected message (assume processes coordinated)
    - if block, process unblocked by NIC on message arrival, and process gets priority boost on wakeup
- Spin-yield (SY)
  - after spinning, process lowers its priority and raises priority of another process (based on pending messages), and keeps spinning
- Demand-based co-scheduling (DCS)
  - incoming message to NIC causes destination process to be scheduled, preempting currently executing process if needed
    - by raising priority of dest. process, so next scheduler invocation will schedule it
- Periodic boost (PB)
  - no interrupt, but kernel periodically checks message queues and boosts priority of 1 process with an outstanding message
  - 10 different heuristics for deciding which process to boost, if multiple ones have messages queued
    - basic one is round-robin across processes with message not consumed, and if there isn't one, round-robin find a process not in a receive call
- Can also combine DCS or PB with SB or SY

Experiments

- Very well done, comprehensive set of experiments
  - fixing all except one parameter for each experiment, doing sensitivity analysis over parameters where that makes sense, etc.
- Overall results show that:
  - DCS is better than gang scheduling, because performs better most of the time, and does not require explicit coordination among processes for scheduling decisions
    - but gang scheduling is more fair across different job types
  - Blocking (SB) better than yielding (SY) most of the time
  - Periodic boost (PB) better than SB (and all other schemes)
    - gets advantages of spinning, and relinquishes CPU when needed
    - PB performance similar to SB for CPU and I/O intensive workloads, PB better for communication intensive ones
    - PB better than SB for smaller multi-programming levels, and for higher system overheads (context switch, interrupts)
    - SB has small edge for high skew (load imbalance)