CMSC 714
Lecture 15
Mtool and Critical Path Profiling

Alan Sussman

Notes

- Midterm exam on November 17
  - sample exam questions posted soon
- Comments on research project proposals by end of day Friday (or earlier)

Mtool

- Tool for performance debugging of parallel applications running on SMPs
- Goal is to figure out where parallel program is spending its time
  - Computation, Synchronization, Memory accesses, doing extra work relative to sequential code
- 2 passes/runs over the (instrumented) code
  - first to do basic block counting, and build an execution profile to determine execution time spent in each b.b.
  - second to pick program sections to instrument to collect times spent there (loops, functions, sync. calls), and basic block counts
- For memory overheads/losses, compare actual times to estimates based on b.b. counts and a perfect memory system (i.e. no cache misses)
- Directly measure sync. overheads
- Extra work comes from parallel control constructs not in sequential code

Mtool (cont.)

- One goal is to minimally perturb performance of the application, so measurements are accurate
- For compute time (needed to measure memory overhead), very simple model of execution of instructions in b.b.
  - likely not at all accurate for current processors
  - but could use modern hardware performance counters
  - for timing, use pc-sampling (low resolution) or direct clock access if that is fast (true on modern processors)
- For sync. overhead, target C with ANL macros and Fortran with parallel DO loops (both similar to OpenMP style parallelization)
  - use combination of instrumentation and pc-sampling for timing
- For parallel overhead, count time in task (thread) spawning, lock allocation, and some cost for each sync. call

Critical Path Profiling

- Goal is to find the longest time-weighted sequence of events in a parallel program
  - including message passing between processes, and synch. ops for shared memory programs, encoded in **Program Activity Graph**
- Basic idea is that only the critical path matters to reduce overall program execution time
  - and iterate if the critical path changes after a program change to reduce the time of some operations on a critical path
- Critical path computed online by piggybacking critical path data (current longest path) onto messages
  - and compute share of critical path for each function selected by user (to reduce cost)
  - insert instrumentation into message send/receive and function entry/exit

Critical Path Profiling

- Critical path zeroing
  - to determine the maximum benefit from improving performance of a function on critical path
  - requires some additional instrumentation added to messages, but then tells you whether it's worth improving a given function
- Overhead from the profiling is not too bad if program doesn't send messages too frequently
- Results show that critical path info is more useful for improving performance than just looking at overall execution time spent in each function
  - but in the cases shown, that's really because the problem is non-parallel sections of code
- For shared memory code, instrument locks and barriers instead of messages
  - but critical path zeroing doesn't work well for programs with non-determinism