

Lecture Topics

- some common misconceptions
- a few useful ideas

Administrivia

- lab 3 description

Some Common Misconceptions

- parallelism is always hard
 - yes, I've seen tiny broken “parallel” codes
 - a few instructions used for years in teaching
 - ten lines of code submitted to peer-reviewed conferences
 - most people who work regularly with parallel code
 - do NOT consider it difficult
 - to write a programs of $O(100)$ lines
 - in practice, people use parallel programming models that
 - make writing $O(100k)$ lines like writing $O(100)$ lines 1000 times
 - without such complexity reduction techniques, you're probably not going to have a working program
- if I work hard enough, the possibilities are endless!
 - Nope.
 - Amdahl's law says...
 - speedup is bounded above by $1 / (\text{sequential fraction})$
 - example
 - if you parallelize code that takes 95% of the time
 - you can't get more than $20\times$ speedup
 - Gustafson's law views another way
 - work scales with usable parallelism
 - was being abused before he formulated it
 - Japanese supercomputers beat Crays on huge dense matrices
 - no application for such large dense matrices existed
- that said,
 - some apps today are missing/simplified due to resource limits
 - become possible/more useful with bigger problem sizes
 - fit evaluation of utility to app, not app to evaluation metric

A Few Useful Ideas

- bulk synchronous paradigm (this style dominates HPC applications)
 - global barriers separate temporal regions of code (called phases)
 - usually $O(100)$ lines long
 - interleaving occurs only within phases
 - similar to Stroustrup’s view of classes’ value [Str, p. 20]:
with classes, programming seems like debugging
lots of little programs rather than one big program
 - does tend to correlate resource usage
 - communicate, barrier
 - compute, barrier
 - repeat
- ways to waste time in parallel
 - “good” reasons
 - push bits around (communicate)
 - do some extra work (to avoid communicating)
 - bicker about priority (contend for shared resources)
 - shared hardware resources
 - shared synchronization variables
 - “bad” reasons
 - twiddle your thumbs (wait for synchronization events)
 - communication completion (send, receive, or both)
 - phase completion (barrier)
 - line up single file (unnecessary serialization)
 - temporally correlated accesses to shared hardware resource
 - coarse synchronization (e.g., one big lock)
- No quotes please! “My professor said communication is a waste of time.”

- load balancing
 - fixed work per thread: simple, but may lead to load imbalance
 - solution: dynamic or partly-dynamic mapping of work to threads
 - one or more queues of tasks
 - pull one or more tasks from a queue, return later for more
 - start with bigger groups, later grab smaller groups
 - if your queue is empty, try to steal work from another one
 - one challenge
 - If new work can be added as a result of executing tasks
 - when are you done? (note: queues are distributed)
 - Rigel Task Model has a nice solution
- many pushes for programmers to produce smaller grain sizes
 - pros
 - more parallelism (scalable)
 - lots of tasks per processor simplifies load balancing (efficient)
 - cons (challenges for parallel languages/models)
 - data locality may be hard to express/exploit
 - more variance, so you need dynamic load balancing
 - splitting natural grain size increases variance
 - microarchitectural affects induce higher variance (one cache miss means more in a short task)
 - sometimes harder to exploit (explicit approaches)
 - more packaging for inter-task communication
 - more dependences
 - more scheduling overhead
 - more communication overhead
 - more complex work distribution (need >1 queue)
 - » contention means more overhead
 - » group tasks dynamically to amortize