

Lecture Topics

- a few useful ideas (wrap-up)
- performance reasoning for shared memory threads (Lab 3)
- POSIX Threads on the Head of a Pin

Administrivia

- lab 3: starting code online

- scheduling parallel jobs
 - time-sharing or space-sharing?
 - sequential OS scheduler is time-sharing
 - if you have several processors, why pay the overhead?
 - allocate blocks of processors to parallel applications
 - leave a couple free for interactive jobs
 - example: Livermore gang scheduling (space-sharing)
 - three classes of jobs
 - batch: get it done before morning
 - compute: I'll be back after coffee
 - interactive: I'm trying to debug!
 - allocate “gangs” of processors to queued apps
 - try to prioritize queue by class
 - example: co-scheduling
 - Ousterhout's Sprite OS at Berkeley, late 1980s
 - coordinate schedulers across distributed workstations
 - start/stop parallel job threads simultaneously
 - overhead high, and outlook bleak for interactive jobs
- fitting into general systems
 - a Glunix result ca. 1995 (global OS for NOW project at Berkeley)
 - for some apps, a few % skew led to 100× slowdown
 - idle waiting for communicated data
 - slow to barrier, but barrier is more messages...
 - a source of skew? Glunix daemons...
 - implicit scheduling (Andrea Arpaci-Dusseau's Ph.D.)
 - use message semantics to adapt local schedule
 - dynamically find co-scheduled solution
- scheduling hierarchical parallelism well?
open problem, I think; see Soumen Chakrabarti's thesis for starter

Shared Memory Thread Performance

- some things you should know, but people often forget
 - parallelizing slow code results in slow parallel code
 - Andrea Arpaci-Dusseau studied parallel sorts in her MS
 - biggest performance impact?
 - performance of threads on individual processors!
 - if 50% of each thread's time is spent in STL...
 - guess what fraction of total time is in STL
 - guess what happens if you parallelize an important STL container with one big lock
 - locality is important, particularly with shared hardware resources
 - shared memory threads compete for resources
 - poor locality can lead to thrashing (performance cliffs)
 - premature optimization is bad...
 - more complexity
 - the fast part is faster...now what?
 - ...usually
 - parallel hardware more likely to vary/change
 - some planning for future hardware may not be a bad idea

- some “free” gains with shared memory threads
 - good management
 - any thread can run on any processor
 - don’t let a processor idle unless there are $< P$ tasks left
 - note: load imbalance is not impossible!
 - bad management
 - don’t let tasks finish
 - interleave them instead
 - more competitors for cache resources
 - more switching overhead
 - migrate regularly
 - bonus communication
 - move data back and forth between caches
 - tension here
 - optimal solution depends on shared resource use
 - data sharing relations between threads
 - synchronization variables
 - microarch. with core multithreading
 - often not easy to understand, let alone express, locality
 - similarly difficult to model cost of migration
 - heuristics try to capture in some models
- synchronization bottlenecks
 - simplicity and relatively low cost of synchronization can be alluring
 - but most synchronization mechanisms designed for low contention
 - performance falls off quickly with higher contention
 - consider
 - how long should a thread wait on a variable before yielding?
 - all of that time is idle time...
 - as is the switch overhead
 - again, not clear that there is a “correct” decision

POSIX Threads on the Head of a Pin

- POSIX error model (must be re-entrant!)
 - return 0 on success
 - return positive error code on failure
- attributes
 - certain aspects of threads must be set at creation time
 - encapsulated in attributes structure: `pthread_attr_t`
 - initialize with `pthread_attr_init`
 - scheduling (`pthread_attr_setscope`)
 - Posix supports threads that run until they yield
 - Linux does not (except for real-time threads)
 - detach/join
 - default behavior is joinable thread
 - thread function returns `void*` value
 - some other thread receives at join point
 - `pthread_join` is blocking
 - avoid need to join with `pthread_attr_setdetachstate`
- creation
 - spawn new thread with

```
int pthread_create (pthread_t* thread,  
                  const pthread_attr_t* attr,  
                  void *(thread_main)(void*), void* arg);
```

 - if successful, new (opaque) thread identifier returned in `*thread`
 - thread main function takes and returns a pointer
 - `arg` is passed as argument to new thread's main function
 - returned value delivered to join call (as above)
- note that termination of main thread immediately ends all others