Performance Considerations in Multi-Threaded Programs

Godmar Back

Virginia Tech

July 12, 2020

Performance Considerations

Upfront Note

Correctness cannot be traded for performance. No one cares about the performance of code that contains data races, atomicity violations, ordering violations, or is prone to deadlocks.

- That said, let's examine the cost of locking in particular
- Indirect cost (resulting in loss of performance due to the use of locking)
  - Simulated on following slides 5 CPU-bound processes contending for L locks, holding each lock for duration D, then running for duration U without lock. Thread chooses lock randomly.
  - Lightgreen are threads running without holding locks
  - Other colors are threads holding locks
- Direct cost (involved in actions the system had to take to implement it)

Indirect Cost: Loss of Parallelism Due To Single Lock

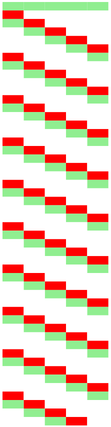

Figure 1: Fixed: U=2/D=2/L=1/40.8%

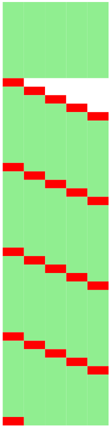

Figure 2: Fixed: U=18/D=2/L=1/96%

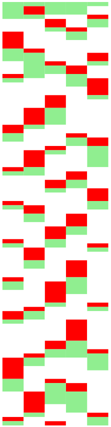

Figure 3: Poisson: U=2/D=2/L=1/41.6%

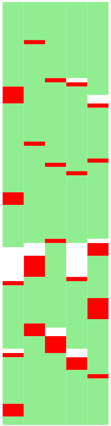

Figure 4: Poisson U=18/D=2/L=1/94.4%

Indirect Cost: Loss of Parallelism with 2 Locks

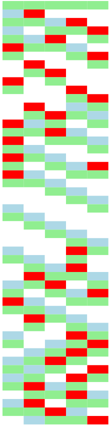

Figure 5: Fixed: U=2/D=2/L=2/65.2%

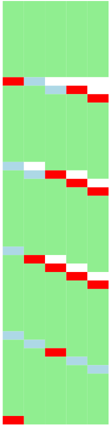

Figure 6: Fixed: U=18/D=2/L=2/96%

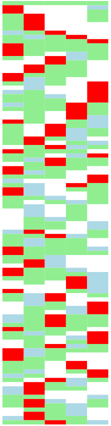

Figure 7: Poisson: U=2/D=2/L=2/72.2%

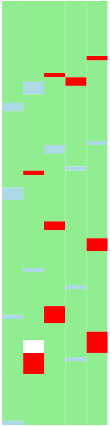

Figure 8: Poisson U=18/D=2/L=2/99.4%

Indirect Cost: Loss of Parallelism with 4 locks

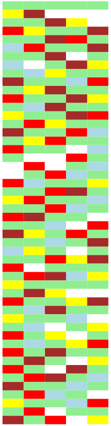

Figure 9: Fixed: U=2/D=2/L=4/89.6%

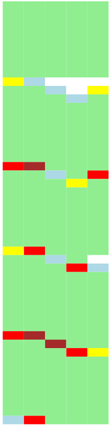

Figure 10: Fixed: U=18/D=2/L=4/98%

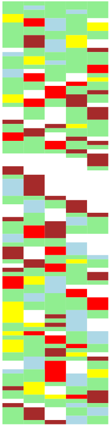

Figure 11: Poisson: U=2/D=2/L=4/79.2%

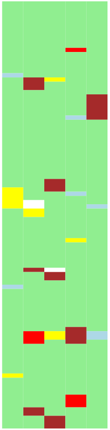

Figure 12: Poisson U=18/D=2/L=4/99.4%

Indirect Cost: Loss of Parallelism

- Serialization due to locks diminishes CPU utilization and increases an individual task's latency
  - For parallel, mostly CPU-bound applications this translates directly into loss of speedup
  - Particularly if locks are contended (situation where threads are blocked on a lock arises frequently)
  - Particularly/assuming if there's nothing else to run during times when threads are blocked
- This serialization effect would be exacerbated if blocked threads held locks (e.g., I/O, sleep, sem_wait, pthread_join?)
- Rule: Critical sections should not call any functions that may block, or else the critical section may become inaccessible

```
pthread_mutex_lock(&shutdownLock);
pthread_mutex_lock(&infoLock);
while (!moreInformation)
    pthread_cond_wait(&moreInfo, &infoLock);
pthread_mutex_unlock(&infoLock);
pthread_mutex_unlock(&shutdownLock);
```

```
pthread_mutex_lock(&lock);
read(fd, buf, sizeof buf);
pthread_mutex_unlock(&lock);
```

VIRGINIA TECH

Solution: Breaking Up Locks

- Cautionary side note: several large software systems were either never parallelized or started with a "big lock" approach: the Linux kernel, Python's GIL, gtk GUI lock
- Idea: instead of having lock L protect data (A, B, C) introduce locks L_A, L_B, L_C to protect A, B, and C, respectively.
- Thus, updates to A will not prevent simultaneous updates to B
- This introduces 3 risks
  1. Higher risk of atomicity violations: if A and B must be updated in tandem (atomically) - say update to B is dependent on A having a value, both locks must be held. Always holding both locks negates purpose of having 2 locks; not holding them both where needed leads to atomicity violations
  2. Higher risk of deadlocks: if there are situations where both locks must be held, a locking order must be established to avoid deadlocks
  3. More frequent calls to lock/unlock translates to increased direct cost (locking overhead)

Direct Cost of Locking

- What happens under the hood in a call to `pthread_mutex_lock()`?
  - Fast path: an atomic instruction tries to acquire the lock (if available) without causing a mode switch (e.g. `cmpxchg %rax, (%rbx)`) - in memory flag that indicates if lock is available
  - For fast path numbers, see Jeff Dean/Peter Norvig/Colin Scott Numbers Every Programmer Should Know: 17× L1 reference, 4× L2 reference, $\frac{1}{6}\times$ main memory reference (17ns as of 2010's)
  - Slow path: if atomic instruction indicates that lock is already held, make system call (`futex_wait`) and inform kernel that thread should block. Then, context switch to other ready thread (if any)
- ... `pthread_mutex_unlock()`?
  - Fast path: just place lock into unlocked state
  - Slow path (someone is waiting for the lock): make system call (`futex_wakeup`) and inform kernel to wake up any waiting thread(s). These threads are unblocked (made ready), placed into ready queue, and eventually scheduled - another context switch
- Both mode and context switches can be costly (e.g. pipeline stalls, cache pollution)

Conclusion

- Optimizing locking is difficult
- Correctness is paramount
- Performance impact can be difficult to predict
- Strategies to reduce serialization may increase locking overhead
- General approach should be start conservatively with coarse-grained locking strategies, and move to finer-grained locking as part of an iterative optimization process