Yes, we need new languages for multicore computing

David Chase

CGO 2007, San Jose, CA
2007-03-12

Abundant, variable parallelism.

- Instead of higher clock rates, more cores
- Think 32-128, not 2-8
- No particular number of threads
  - Lost to chip flaws
  - Lost to other bottlenecks (L2 cache)
  - Lost to other processes
- Workstealing is very effective; on-chip locality is good enough

Current popular languages micromanage execution

- Parallel only when specified
- but mandatory parallelism when specified
  - heavyweight threads
  - exactly N threads
- Cannot say “I don’t care”
- Need more implicit parallelism
  - Loops
  - Function and operator inputs

Need transactions instead of locks

- “Locks don’t compose”
- Locks are too hard for programmers, even with today’s limited parallelism
- Deadlocks and bottlenecks scale non-linearly
- Locks are pessimistic and impede parallelism
- Little hope of understanding lock orders in a world with implicit parallelism

Must have a memory model and programmers must learn it.

```
SomeClass sharedThing; /* Should be volatile */

SomeClass getSharedThing() {
    if (sharedThing == null)
        synchronized (this) {
            if (sharedThing == null) {
                sharedThing = initialValue();
                /* Other threads may see non-null
                   sharedThing, but stores from
                   initialValue may not be flushed
                 */
            }
        } /* Synchronized memory barrier here */
    return sharedThing;
}
```

Side-effects should be unusual

- The Java Programming Language™, C, C++ -- mutable fields are the default case.
  Immutable would be better for parallelism.
  - Tool enabler
  - Optimizer can work more locally
- Java Collections API -- all mutable; need immutable variants.
- Applicative data structures are not necessarily any slower (in one real test, 20% faster on a uniprocessor)

Must have garbage collection

- Applicative data structures are difficult to manage
- Manual memory management in parallel is tricky and often slow (e.g., consistent reference counting)
- GC is generally helpful
- GC simplifies tricky concurrent algorithms
- Lots of synergy between GC and transactions; the cost is subadditive, you might as well enjoy the benefits.

Yes, we need new languages for multicore computing

david.chase@sun.com