Benchmarking Concurrent Code

Go Benchmark Basics

```
func BenchmarkFunction(b *testing.B) {
    for i := 0; i < b.N; i++ {
        function()
    }
}

// Run: go test -bench=.
// Output: BenchmarkFunction-8  1000000  1234 ns/op
//         Name                 cpus    iterations  time/op
```

Key metrics:

- **ns/op:** Time per operation
- **B/op:** Bytes allocated per operation
- **allocs/op:** Allocations per operation

Benchmarking Concurrent Operations

```
func BenchmarkConcurrentCounter(b *testing.B) {
    var counter atomic.Int64

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            counter.Add(1)
        }
    })
}

// RunParallel: Runs function in GOMAXPROCS goroutines
// pb.Next(): Returns true while benchmark running
```

Comparing Mutex vs. Atomic

```
func BenchmarkMutexIncrement(b *testing.B) {
    var (
        counter int
        mu      sync.Mutex
    )

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            mu.Lock()
            counter++
            mu.Unlock()
```

```
        }
    })
}

func BenchmarkAtomicIncrement(b *testing.B) {
    var counter atomic.Int64

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            counter.Add(1)
        }
    })
}

// Results (typical on 8-core machine):
// BenchmarkMutexIncrement-8     20000000    85 ns/op
// BenchmarkAtomicIncrement-8    50000000    30 ns/op
//
// Atomic is ~3x faster for simple increment
```

Benchmarking Channels

```
func BenchmarkUnbufferedChannel(b *testing.B) {
    ch := make(chan int)

    go func() {
        for range ch {
        }
    }()

    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        ch <- i
    }

    close(ch)
}

func BenchmarkBufferedChannel(b *testing.B) {
    ch := make(chan int, 100)

    go func() {
        for range ch {
        }
    }()

    b.ResetTimer()

    for i := 0; i < b.N; i++ {
```

```
        ch <- i
    }

    close(ch)
}

// Results:
// BenchmarkUnbufferedChannel-8    5000000    312 ns/op
// BenchmarkBufferedChannel-8     20000000     65 ns/op
//
// Buffered channels ~5x faster (less synchronization)
```

Benchmarking Different Concurrency Levels

```
func BenchmarkWorkerPool(b *testing.B) {
    workerCounts := []int{1, 2, 4, 8, 16, 32}

    for _, workers := range workerCounts {
        b.Run(fmt.Sprintf("workers=%d", workers), func(b *testing.B) {
            pool := NewWorkerPool(workers)
            defer pool.Shutdown()

            b.ResetTimer()

            for i := 0; i < b.N; i++ {
                pool.Submit(func() {
                    doWork()
                })
            }

            pool.Wait()
        })
    }
}

// Output:
// BenchmarkWorkerPool/workers=1-8      100000   15234 ns/op
// BenchmarkWorkerPool/workers=2-8      200000    7891 ns/op
// BenchmarkWorkerPool/workers=4-8      400000    4123 ns/op
// BenchmarkWorkerPool/workers=8-8      500000    3456 ns/op
// BenchmarkWorkerPool/workers=16-8     450000    3789 ns/op
// BenchmarkWorkerPool/workers=32-8     400000    4234 ns/op
//
// Optimal: 8 workers (matches CPU cores)
```

Memory Allocation Benchmarks

```go
func BenchmarkMapWithMutex(b *testing.B) {
    m := make(map[string]int)
    var mu sync.RWMutex

    b.ReportAllocs()
    b.ResetTimer()

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            mu.Lock()
            m["key"] = 42
            mu.Unlock()
        }
    })
}

func BenchmarkSyncMap(b *testing.B) {
    var m sync.Map

    b.ReportAllocs()
    b.ResetTimer()

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            m.Store("key", 42)
        }
    })
}

// Run: go test -bench=. -benchmem
//
// BenchmarkMapWithMutex-8   10000000   145 ns/op   0 B/op   0 allocs/op
// BenchmarkSyncMap-8        20000000    75 ns/op   8 B/op   1 allocs/op
//
// sync.Map has allocation overhead but faster under contention
```

Benchmarking Lock Contention

```go
func BenchmarkLockContention(b *testing.B) {
    contentionLevels := []int{1, 10, 100}

    for _, workDuration := range contentionLevels {
        b.Run(fmt.Sprintf("work=%dns", workDuration), func(b *testing.B) {
            var (
                counter int
                mu      sync.Mutex
            )

            b.RunParallel(func(pb *testing.PB) {
```

```
                for pb.Next() {
                    mu.Lock()

                    // Simulate work under lock
                    start := time.Now()
                    for time.Since(start) < time.Duration(workDuration) {
                    }

                    counter++
                    mu.Unlock()
                }
            })
        })
    }
}

// Output shows contention impact:
// BenchmarkLockContention/work=1ns-8      50000000    35 ns/op
// BenchmarkLockContention/work=10ns-8     30000000    45 ns/op
// BenchmarkLockContention/work=100ns-8    10000000   145 ns/op
//
// Longer critical sections → more contention → slower
```

Comparing Synchronization Primitives

```
func BenchmarkMutex(b *testing.B) {
    var (
        data int
        mu   sync.Mutex
    )

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            mu.Lock()
            data++
            mu.Unlock()
        }
    })
}

func BenchmarkRWMutex(b *testing.B) {
    var (
        data int
        mu   sync.RWMutex
    )

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            mu.Lock()
            data++
```

```
            mu.Unlock()
        }
    })
}

func BenchmarkAtomic(b *testing.B) {
    var data atomic.Int64

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            data.Add(1)
        }
    })
}

func BenchmarkChannel(b *testing.B) {
    ch := make(chan int, 100)

    go func() {
        for range ch {
        }
    }()

    b.ResetTimer()

    for i := 0; i < b.N; i++ {
        ch <- 1
    }

    close(ch)
}

// Results (typical):
// BenchmarkMutex-8      20000000    85 ns/op
// BenchmarkRWMutex-8    15000000   110 ns/op  (slower for writes)
// BenchmarkAtomic-8     50000000    30 ns/op  (fastest)
// BenchmarkChannel-8    10000000   165 ns/op  (slowest, most flexible)
```

Read-Heavy Workload Benchmark

```
func BenchmarkRWMutexReadHeavy(b *testing.B) {
    var (
        data int
        mu   sync.RWMutex
    )

    // 90% reads, 10% writes
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            if rand.Intn(10) == 0 {
```

```
                // Write
                mu.Lock()
                data++
                mu.Unlock()
            } else {
                // Read
                mu.RLock()
                _ = data
                mu.RUnlock()
            }
        }
    })
}

func BenchmarkMutexReadHeavy(b *testing.B) {
    var (
        data int
        mu   sync.Mutex
    )

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            if rand.Intn(10) == 0 {
                mu.Lock()
                data++
                mu.Unlock()
            } else {
                mu.Lock()
                _ = data
                mu.Unlock()
            }
        }
    })
}

// Results for read-heavy:
// BenchmarkRWMutexReadHeavy-8   50000000    35 ns/op  (wins!)
// BenchmarkMutexReadHeavy-8     20000000    85 ns/op
//
// RWMutex better when reads >> writes
```

Benchmarking with Profiling

```
# CPU profile
go test -bench=. -cpuprofile=cpu.prof
go tool pprof cpu.prof

# Memory profile
go test -bench=. -memprofile=mem.prof
go tool pprof mem.prof
```

```
# Blocking profile
go test -bench=. -blockprofile=block.prof
go tool pprof block.prof
```

Interpreting Benchmark Results

```
// Example output:
// BenchmarkFunction-8    1000000    1234 ns/op    456 B/op    7 allocs/op

// -8: GOMAXPROCS (CPU cores used)
// 1000000: Number of iterations
// 1234 ns/op: Time per operation
// 456 B/op: Bytes allocated per operation
// 7 allocs/op: Number of allocations per operation
```

Lower is better for all metrics!

Benchmark Comparison Tool

```
# Run baseline
go test -bench=. > old.txt

# Make changes

# Run new benchmark
go test -bench=. > new.txt

# Compare
go install golang.org/x/perf/cmd/benchstat@latest
benchstat old.txt new.txt

# Output:
# name         old time/op  new time/op  delta
# Function-8     1.23µs ± 2%    0.85µs ± 1%  -30.89%  (p=0.000 n=10+10)
```

CI/CD Benchmark Regression Detection

```
// Test: Benchmarks must not regress
func TestBenchmarkRegression(t *testing.T) {
    if testing.Short() {
        t.Skip()
    }

    // Run benchmark
    result := testing.Benchmark(BenchmarkFunction)
```

```
    // Check against baseline (e.g., 1000 ns/op)
    baseline := int64(1000)
    threshold := baseline * 110 / 100  // Allow 10% regression

    nsPerOp := result.NsPerOp()
    if nsPerOp > threshold {
        t.Errorf("Benchmark regressed: %d ns/op (threshold: %d ns/op)",
            nsPerOp, threshold)
    }
}
```

Real-World Example: Cache Benchmark

```
type Cache interface {
    Get(key string) (interface{}, bool)
    Set(key string, value interface{})
}

func BenchmarkCache(b *testing.B) {
    caches := map[string]Cache{
        "mutex": NewMutexCache(),
        "sync.Map": NewSyncMapCache(),
        "sharded": NewShardedCache(16),
    }

    for name, cache := range caches {
        b.Run(name, func(b *testing.B) {
            // Warm up
            for i := 0; i < 1000; i++ {
                cache.Set(fmt.Sprint(i), i)
            }

            b.ResetTimer()
            b.ReportAllocs()

            b.RunParallel(func(pb *testing.PB) {
                i := 0
                for pb.Next() {
                    key := fmt.Sprint(i % 1000)
                    if i%10 == 0 {
                        cache.Set(key, i)
                    } else {
                        cache.Get(key)
                    }
                    i++
                }
            })
        })
    }
}
```

```
// Output:
// BenchmarkCache/mutex-8         5000000    345 ns/op    32 B/op   2 allocs/op
// BenchmarkCache/sync.Map-8     10000000    156 ns/op    48 B/op   3 allocs/op
// BenchmarkCache/sharded-8      15000000     98 ns/op    32 B/op   2 allocs/op
//
// Sharded cache wins! (reduces lock contention)
```

Sub-benchmarks for Different Scenarios

```
func BenchmarkWorkerPool(b *testing.B) {
    taskDurations := []time.Duration{
        0,                    // No work
        10 * time.Microsecond,   // Light
        100 * time.Microsecond,  // Medium
        time.Millisecond,     // Heavy
    }

    for _, duration := range taskDurations {
        b.Run(fmt.Sprintf("task=%v", duration), func(b *testing.B) {
            pool := NewWorkerPool(runtime.NumCPU())
            defer pool.Shutdown()

            b.ResetTimer()

            for i := 0; i < b.N; i++ {
                pool.Submit(func() {
                    time.Sleep(duration)
                })
            }

            pool.Wait()
        })
    }
}
```

Common Mistakes in Benchmarks

Mistake #1: Not Resetting Timer

```
// WRONG: Setup time included in benchmark
func BenchmarkWrong(b *testing.B) {
    cache := setupExpensiveCache()  // Counted!

    for i := 0; i < b.N; i++ {
        cache.Get("key")
    }
}
```

```
// Right: Reset timer after setup
func BenchmarkRight(b *testing.B) {
    cache := setupExpensiveCache()
    b.ResetTimer()  // Reset!

    for i := 0; i < b.N; i++ {
        cache.Get("key")
    }
}
```

Mistake #2: Compiler Optimization Removing Code

```
// WRONG: Compiler may optimize away unused result
func BenchmarkWrong(b *testing.B) {
    for i := 0; i < b.N; i++ {
        compute()  // Result unused, might be optimized out
    }
}

// Right: Store result to prevent optimization
var result int

func BenchmarkRight(b *testing.B) {
    var r int
    for i := 0; i < b.N; i++ {
        r = compute()
    }
    result = r  // Assign to package-level var
}
```

Mistake #3: Benchmarking Wrong Thing

```
// WRONG: Benchmarks goroutine creation, not actual work
func BenchmarkWrong(b *testing.B) {
    for i := 0; i < b.N; i++ {
        go doWork()  // Just spawns goroutine
    }
}

// Right: Wait for work completion
func BenchmarkRight(b *testing.B) {
    for i := 0; i < b.N; i++ {
        done := make(chan struct{})
        go func() {
            doWork()
            close(done)
        }()
        <-done
```

```
    }
}
```

Interview Questions

Q: "How do you benchmark concurrent code?"

"Use b.RunParallel() which runs benchmark function in GOMAXPROCS goroutines. Pattern: b.RunParallel(func(pb *testing.PB) { for pb.Next() { operation() } }). Measures real concurrent performance with contention. Also benchmark with different concurrency levels using sub-benchmarks to find optimal worker count."

Q: "What's the difference between ns/op, B/op, and allocs/op?"

"ns/op: nanoseconds per operation, measures time/performance. B/op: bytes allocated per operation, measures memory usage. allocs/op: number of allocations per operation, measures GC pressure. Lower is better for all. Optimize in order: correctness → performance (ns/op) → memory (B/op) → GC pressure (allocs/op)."

Q: "How do you detect performance regressions?"

"Use benchstat to compare old vs new benchmarks: go test -bench=. > old.txt, make changes, go test -bench=. > new.txt, benchstat old.txt new.txt. Shows statistical significance. In CI: Run benchmark in tests, fail if > threshold (e.g., 10% regression). Store baseline, alert on regression."

Q: "When should you use RWMutex vs Mutex?"

"RWMutex better when reads >> writes (>5:1 ratio). Benchmark both: Mutex has lower overhead for writes, RWMutex allows concurrent reads. Pattern: Benchmark with realistic read/write ratio. If 90% reads, RWMutex wins. If 50/50, Mutex often faster. Always measure, don't guess."

Key Takeaways

1. **Use b.RunParallel() for concurrent benchmarks**
2. **Always b.ResetTimer() after setup**
3. **Assign results to prevent compiler optimization**
4. **Use -benchmem to measure allocations**
5. **Compare with benchstat for statistical significance**
6. **Benchmark different concurrency levels**
7. **Profile with -cpuprofile, -memprofile**
8. **Lock contention kills performance**
9. **Atomic fastest, channels slowest, mutexes middle**
10. **Always benchmark, don't assume**

Exercises

1. Benchmark: Compare atomic, mutex, RWMutex, channel for counter.
2. Find optimal worker count: Benchmark 1, 2, 4, 8, 16, 32 workers.
3. Benchmark your cache: Test mutex, sync.Map, sharded approaches.
4. Measure lock contention: Vary critical section duration, plot results.

5. Use benchstat: Make optimization, prove it's better statistically.

Next: tracing.md - Using Go's execution tracer to visualize concurrency.