

Lösung: ConBench – Multithreaded Benchmarks

runBenchmark

```
public void runBenchmark(BenchmarkDescriptor desc) {
    try {
        println(desc.testClass.getSimpleName() + ": Warming up ...");
        warmup(desc); // (1)
        println(" Starting benchmark ...");
        for (int i = 0; i < desc.testMethods.get(0).nThreads.length; i++) {
            runCombination(desc, i); // (2)
        }
        println("");
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Diese Methode ist der Haupteinstiegspunkt, der vom Interface vorgegeben ist. Vor den aktuellen Messungen wird ein JVM Warmup durchgeführt (1). Dann werden die Messungen gestartet. Jede @Contention Methode hat einen gleich langen int[], dessen Länge die Anzahl durchzuführender Messungen definiert. Entsprechend wird die for-Schleife desc.testMethods.get(0).nThreads.length mal ausgeführt. Jede Konfiguration wird dann separat gemessen (2).

warmup

```
private void warmup(BenchmarkDescriptor desc) throws Exception {
    final Object testObject = desc.testClass.newInstance(); // (1)

    for (ContentionDescriptor cd: desc.testMethods) {
        cd.method.invoke(testObject, desc.nTimes, 1); // (2)
    }
}
```

Zuerst wird ein neues Objekt der @Benchmark annotierten Klasse erzeugt (1). Falls Sie Initialisierungen im Konstruktor machen wollen, macht es Sinn für den Warmup und für die Messung separate Objekte zu erstellen. Dann wird jede Methode desc.nTimes mal aufgerufen (2). Da Sie nicht wissen, wie Zeitaufwändig eine Messung ist, wäre ungeschickt da einfach einen hohen Wert zu verwenden.

runCombination

```
private void runCombination(final BenchmarkDescriptor desc, int comb)
    throws Exception {

    String run = " - Run[" + comb + "]" ;

    int nThreads = 0;
    for (ContentionDescriptor tcd: desc.testMethods) { // (1)
        nThreads += tcd.nThreads[comb];
    }

    final TimerAction timer = new TimerAction(); // (2)
    final CyclicBarrier barrier = new CyclicBarrier(nThreads + 1, timer); // (3)

    final Object testObject = desc.testClass.newInstance(); // (4)
```

```
// for all test methods
for(final ContentionDescriptor tcd: desc.testMethods) { // (5)
    String methName = tcd.method.getName();
    run += " " + methName + "(" + tcd.nThreads[comb] + "),";
    final int nThreadsPerMethod = tcd.nThreads[comb];

    for (int i = 0; i < nThreadsPerMethod; i++) { // (6)
        Thread t = new Thread() {
            public void run() {
                try {
                    barrier.await(); // (7)
                    tcd.method.invoke(testObject, desc.nTimes,
                                    nThreadsPerMethod); // (8)
                    barrier.await(); // (9)
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
            }
        };
        t.setName(methName + " - " + i);
        t.start(); // (10)
    }
}

barrier.await(); // (11) start: wait for test threads
barrier.await(); // (12) end: wait for test threads

long duration = timer.duration() / desc.nTimes; // (13)
println(run + " Duration: " + scaleTime(duration)); // (14)
}
```

Zuerst wird die Anzahl der notwendigen Threads aufaddiert (1). Dazu wird für jeden Contention-Descriptor der entsprechende Wert aus dem `int[]` gelesen. Dann wird eine `TimerAction` Instanz erzeugt (2). Dieses Objekt wird später die Zeiten messen. Sie wird im Anschluss noch kurz vorgestellt. Dann wird eine `CyclicBarrier` erzeugt mit den notwendigen Threads + 1 und der erstellten `TimerAction` (3). Der zusätzlich Thread (+1) wird verwendet um den Ablauf zu koordinieren. Dann wird das Objekt erzeugt (4) auf dem dann die Methoden ausgeführt werden.

Für jede zu testende Methode (5) und die Anzahl paralleler Threads (6) wird ein neuer Thread erzeugt. Jeder dieser Threads wartet als erstes in seiner `run` Methode auf der `Barrier` (7). Dann führt er seine Testmethode aus (8). Wenn diese terminiert, wird wieder auf der `Barrier` gewartet (9) bis alle anderen Threads auch soweit sind. Der erstellte Thread wird dann gleich gestartet (10).

Die Test-Threads sind jetzt alle bereit und warten an der `Barrier`. Der `main`-Thread startet nun den Vorgang, indem er auch ein `await()` aufruft (dafür war das + 1) (11). Dabei nimmt die `TimerAction` die Startzeit. Gleich danach wartet der `main`-Thread bis alle Test-Threads fertig sind (12). Wenn die `Barrier` zum zweiten Mal durchbrochen wird, zeichnet die `TimerAction` die Endzeit auf. Die gemessene Zeit noch durch `nTimes` geteilt (13) und skaliert ausgegeben (14).

TimerAction

```
final class TimerAction implements Runnable {  
    volatile long startTime = 0;  
    volatile long endTime = 0;  
  
    @Override  
    public void run() {  
        if (startTime == 0) {  
            startTime = System.nanoTime();  
        } else {  
            endTime = System.nanoTime();  
        }  
    }  
  
    public long duration() {  
        return endTime - startTime;  
    }  
}
```

Die TimerAction ist ein Runnable, das beim ersten Aufruf der run Methode die Startzeit und beim zweiten Aufruf die Endzeit setzt.