Java Threads

Tomasz Mozolewski
mozotom@yahoo.com

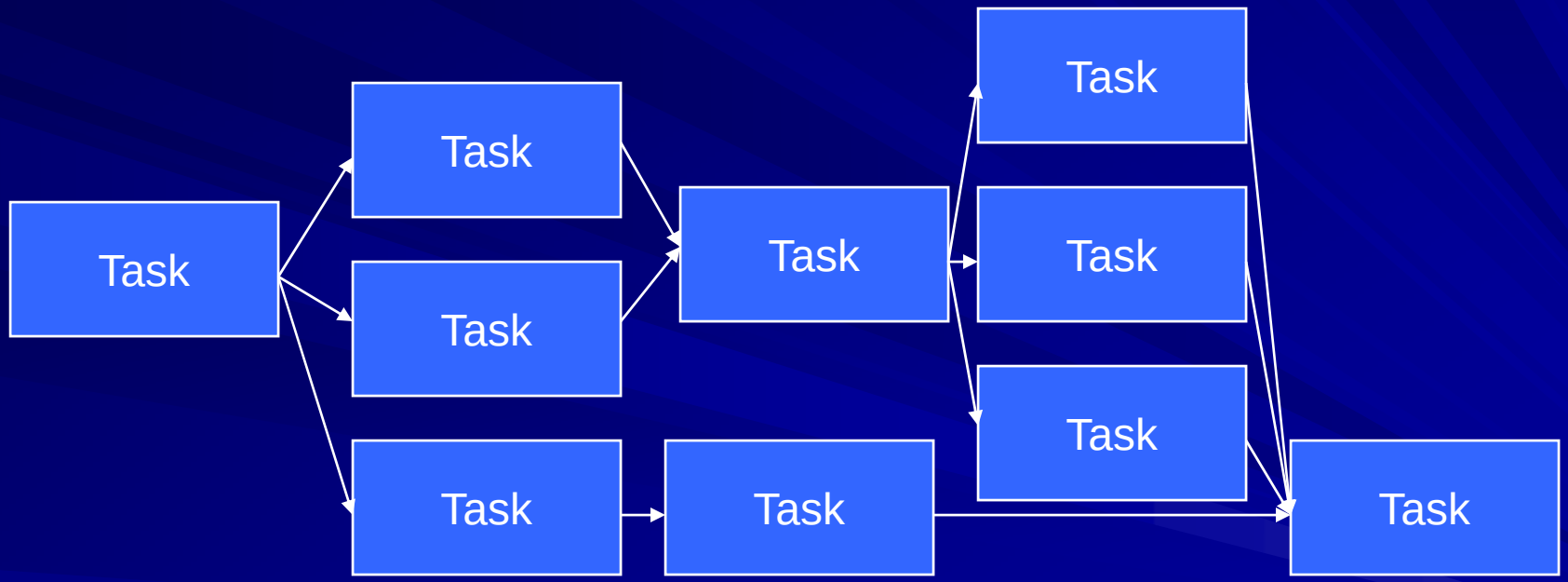
What is this presentation about

- Why parallel programming
- CPU - cache - memory
- Java Memory Model JSR-133
- What does Java offer?
- Cost of various options
- Common pitfalls

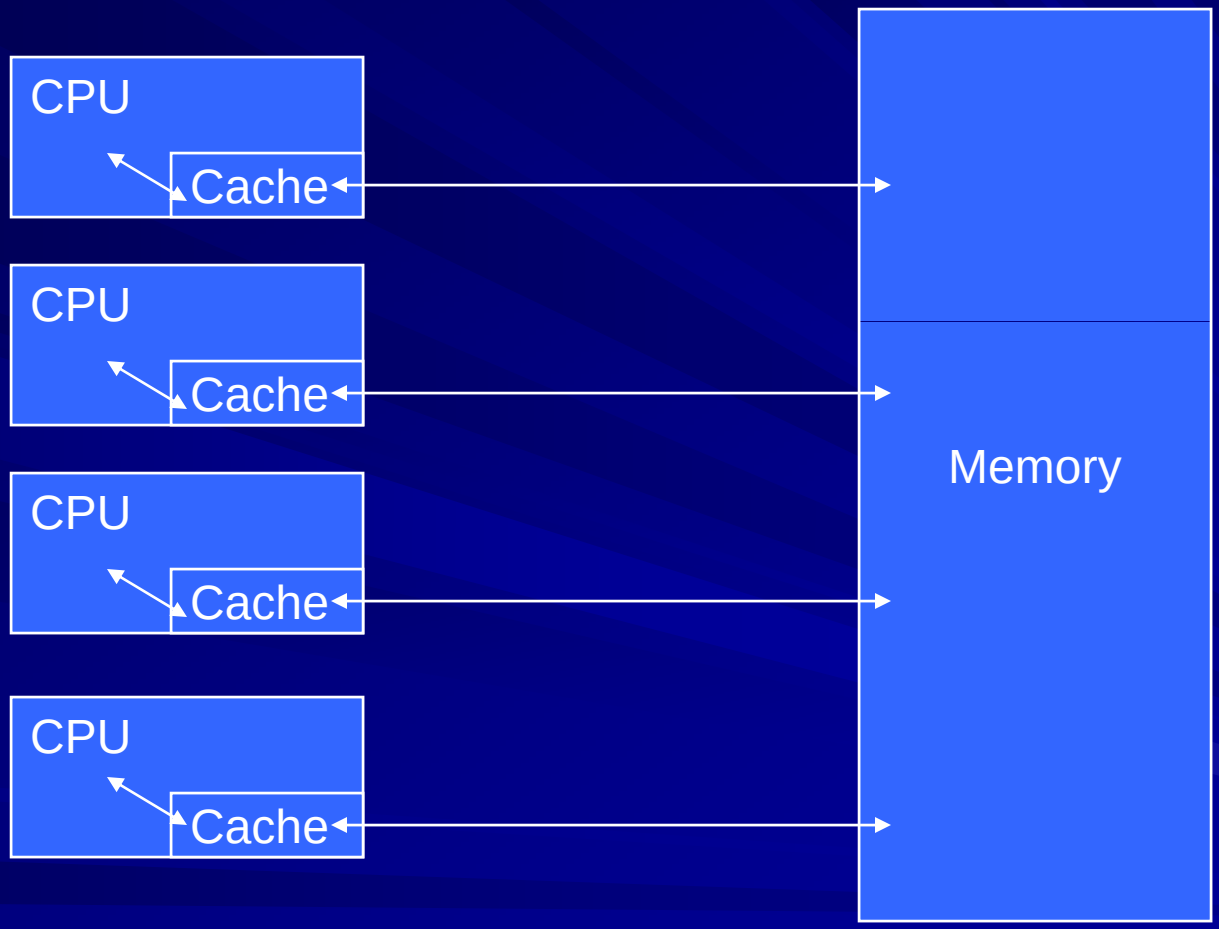
Why parallel programming

- Limit on CPU speed
- Multi-core CPUs, multi CPU systems
- Need to process multiple requests simultaneously (application servers)
- Need to process large amount of data (Monte Carlo simulations)
- Need for faster response

Program execution graph



CPU – Cache - Memory



Java Memory Model JSR-133

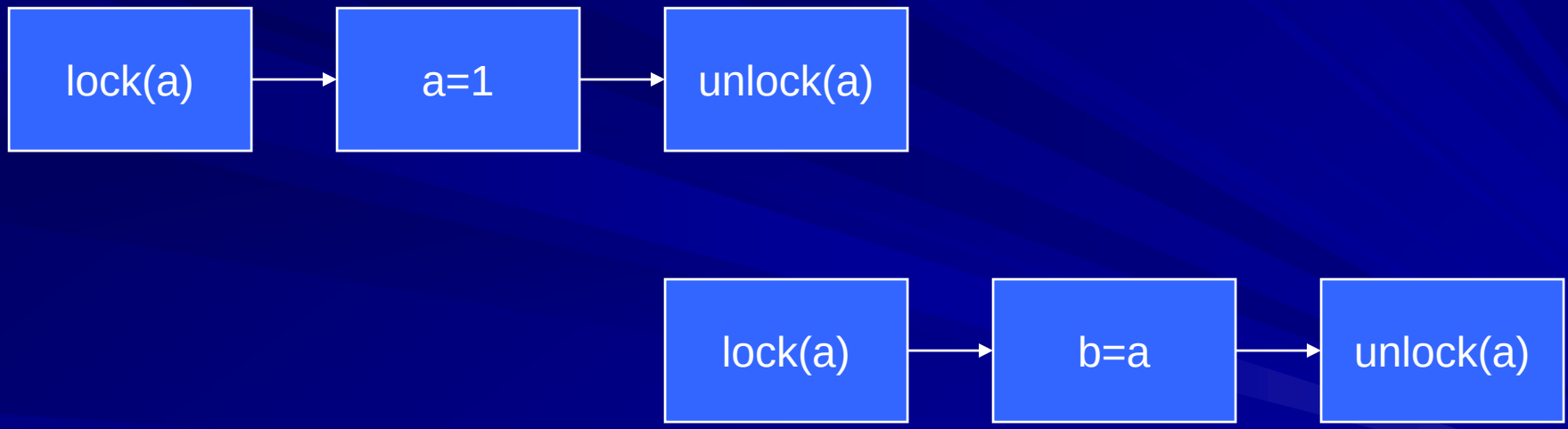
- Compiler reordering
- Cache ⇔ Memory synchronization
- Happens-before relationship

Happens-Before 1 of 6

- Each action in a thread happens before every subsequent action in that thread.

Happens-Before 2 of 6

- An unlock on a monitor happens before every subsequent lock on that monitor.



Happens-Before 3 of 6

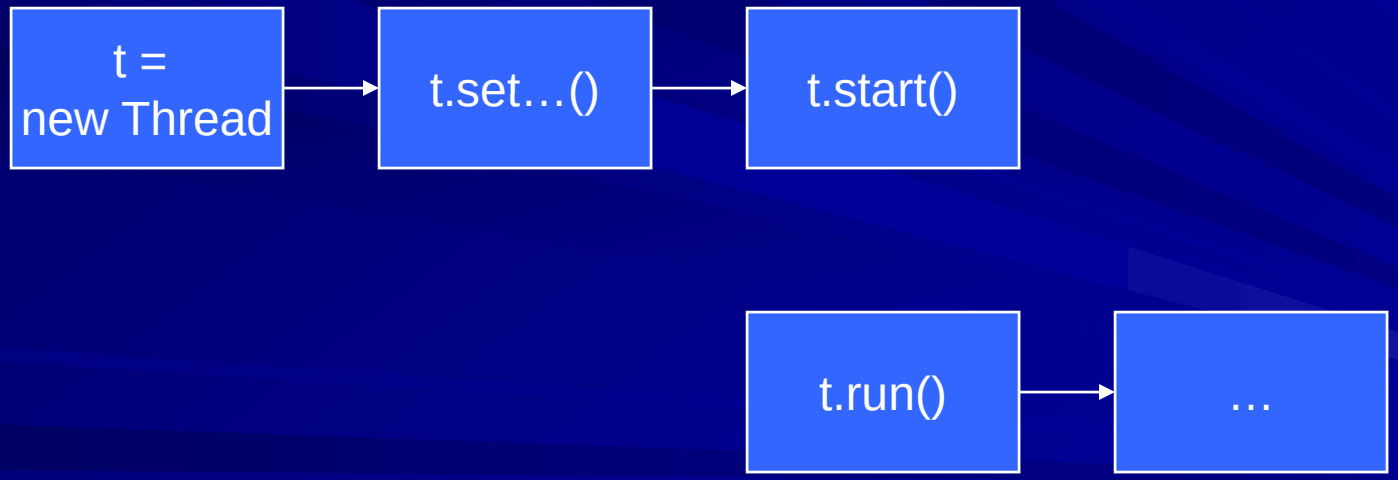
- A write to a volatile field happens before every subsequent read of that volatile.

```java
package com.tomaszmozolewski.examples.concurency.jmm;

public class DangerousLoop implements RunnableLoop {
    boolean keepRunning = true;

    @Override
    public void run() {
        while (keepRunning) Thread.yield();
    }

    public void setKeepRunning(boolean keepRunning) {
        this.keepRunning = keepRunning;
    }
}
```

```java
package com.tomaszmozolewski.examples.concurency.jmm;

public class VolatileLoop implements RunnableLoop {
    volatile boolean keepRunning = true;

    @Override
    public void run() {
        while (keepRunning) Thread.yield();
    }

    public void setKeepRunning(boolean keepRunning) {
        this.keepRunning = keepRunning;
    }
}
```

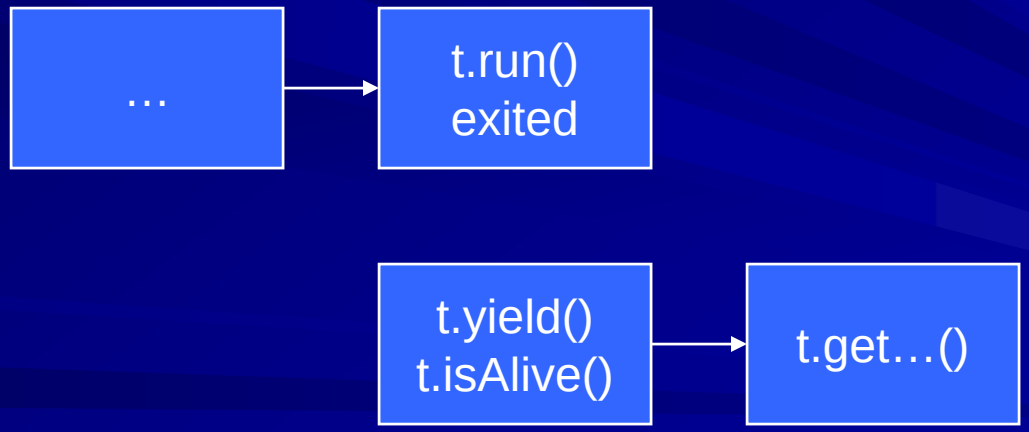
Happens-Before 4 of 6

- A call to start() on a thread happens before any actions in the started thread.



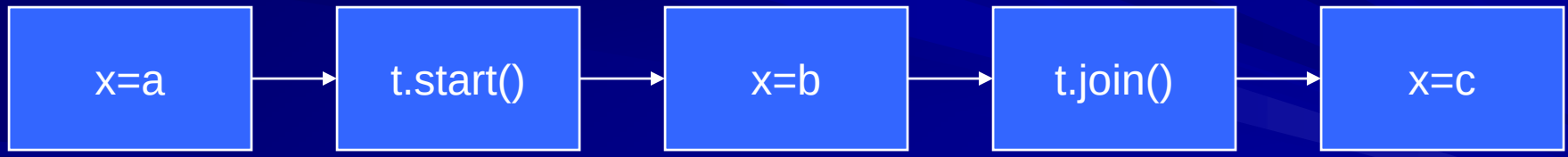
Happens-Before 5 of 6

- All actions in a thread happen before any other thread successfully returns from a join() on that thread.



Happens-Before 6 of 6

- If an action a happens before an action b, and b happens before an action c, then a happens before c.



What does java offer?

- Threads
- Synchronization (locks)
- Wait / notify
- Synchronized collections
- Concurrent collections
- Atomic classes
- Immutable (final) attributes
- Blocking queues
- Executors

Performance

- Synchronization
- Resource contention
- Cost of thread creation
- Serial vs. parallel execution

Counter

```
 1 package com.tomaszmozolewski.examples.concurency.counter;
 2
 3 public interface Counter {
 4
 5     public void inc();
 6
 7     public void dec();
 8
 9     public int getValue();
10 }
```

Counter Thread

```
 1 package com.tomaszmozolewski.examples.concurency.counter;
 2
 3 public class CounterThread implements Runnable {
 4     Counter counter;
 5     int count;
 6     boolean isInc;
 7
 8     public CounterThread(Counter counter, int count, boolean isInc) {
 9         this.counter = counter;
10         this.count = count;
11         this.isInc = isInc;
12     }
13
14     public void run() {
15         if (this.isInc) {
16             while (this.count-->0) this.counter.inc();
17         } else {
18             while (this.count-->0) this.counter.dec();
19         }
20     }
21 }
```

Test Parameters

```
 1 package com.tomaszmozolewski.examples.concurency.counter;
 2
 3 import org.apache.commons.logging.Log;
 4 import org.apache.commons.logging.LogFactory;
 5 import org.junit.Test;
 6
 7 public class CounterTest {
 8     private final static Log log = LogFactory.getLog(CounterTest.class);
 9     final static int THREAD_COUNT = 4;
10     final static int COUNT_TO = 30000000;
```

Basic Counter Implementation

```
1  package com.tomaszmozolewski.examples.concurency.counter;
2
3  public class CounterImpl implements Counter {
4      private int value = 0;
5
6      public void inc() {
7          ++this.value;
8      }
9
10     public void dec() {
11         --this.value;
12     }
13
14     public int getValue() {
15         return this.value;
16     }
17 }
```

Serial Test

```
@Test
public void serialTest() throws InterruptedException {
    Counter counter = new CounterImpl();
    Thread[] threads = new Thread[THREAD_COUNT];

    for (int i=0; i<threads.length; ++i) {
        threads[i] = new Thread(new CounterThread(counter, COUNT_TO, i%2==0));
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.run();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Serial - Counter value: " + counter.getValue() + " Time: " + (
    }
}
```

INFO: Serial - Counter value: 0 Time: 313 ms.

Basic Counter Test

```
@Test
public void counterImplTest() throws InterruptedException {
    Counter counter = new CounterImpl();
    Thread[] threads = new Thread[THREAD_COUNT];

    for (int i=0; i<threads.length; ++i) {
        threads[i] = new Thread(new CounterThread(counter, COUNT_TO, i%2==0));
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.start();
    for (Thread thread: threads) thread.join();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Basic Impl - Counter value: " + counter.getValue() + " Time: "
    }
}
```

INFO: Basic Impl - Counter value: 4801878 Time: 625 ms.

Volatile Counter

```
1  package com.tomaszmozolewski.examples.concurency.counter;
2
3  public class VolatileCounter implements Counter {
4      private volatile int value = 0;
5
6      public void inc() {
7          ++this.value;
8      }
9
10     public void dec() {
11         --this.value;
12     }
13
14     public int getValue() {
15         return this.value;
16     }
17 }
```

Volatile Counter Test

```
@Test
public void volatileCounterTest() throws InterruptedException {
    Counter counter = new VolatileCounter();
    Thread[] threads = new Thread[THREAD_COUNT];

    for (int i=0; i<threads.length; ++i) {
        threads[i] = new Thread(new CounterThread(counter, COUNT_TO, i%2==0));
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.start();
    for (Thread thread: threads) thread.join();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Volatile - Counter value: " + counter.getValue() + " Time: " +
    }
}
```

INFO: Volatile - Counter value: 1064404 Time: 10422 ms

Synchronized Counter

```
1  package com.tomaszmozolewski.examples.concurency.counter;
2
3  public class SyncCounter implements Counter {
4      private int value = 0;
5
6      public synchronized void inc() {
7          ++this.value;
8      }
9
10     public synchronized void dec() {
11         --this.value;
12     }
13
14     public synchronized int getValue() {
15         return this.value;
16     }
17 }
```

Synchronized Counter Test

```
@Test
public void syncCounterTest() throws InterruptedException {
    Counter counter = new SyncCounter();
    Thread[] threads = new Thread[THREAD_COUNT];

    for (int i=0; i<threads.length; ++i) {
        threads[i] = new Thread(new CounterThread(counter, COUNT_TO, i%2==0));
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.start();
    for (Thread thread: threads) thread.join();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Sync - Counter value: " + counter.getValue() + " Time: " + (t
    }
}
```

INFO: Sync - Counter value: 0 Time: 69219 ms.

Atomic Counter

```
1  package com.tomaszmozolewski.examples.concurency.counter;
2
3  import java.util.concurrent.atomic.AtomicInteger;
4
5  public class AtomicCounter implements Counter {
6      private AtomicInteger value = new AtomicInteger(0);
7
8      public void inc() {
9          this.value.incrementAndGet();
10     }
11
12     public void dec() {
13         this.value.decrementAndGet();
14     }
15
16     public int getValue() {
17         return this.value.get();
18     }
19 }
```

Atomic Counter Test

```
@Test
public void atomicCounterTest() throws InterruptedException {
    Counter counter = new AtomicCounter();
    Thread[] threads = new Thread[THREAD_COUNT];

    for (int i=0; i<threads.length; ++i) {
        threads[i] = new Thread(new CounterThread(counter, COUNT_TO, i%2==0));
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.start();
    for (Thread thread: threads) thread.join();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Atomic - Counter value: " + counter.getValue() + " Time: " +
    }
}
```

INFO: Atomic - Counter value: 0 Time: 33750 ms.

Serial Volatile Test

```
@Test
public void serialVolatileTest() throws InterruptedException {
    Counter counter = new VolatileCounter();
    Thread[] threads = new Thread[THREAD_COUNT];

    for (int i=0; i<threads.length; ++i) {
        threads[i] = new Thread(new CounterThread(counter, COUNT_TO, i%2==0));
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.run();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Serial volatile - Counter value: " + counter.getValue() + " Ti
    }
}
```

INFO: Serial volatile - Counter value: 0 Time: 1172 ms.

Serial Sync Test

```
@Test
public void serialSyncTest() throws InterruptedException {
    Counter counter = new SyncCounter();
    Thread[] threads = new Thread[THREAD_COUNT];

    for (int i=0; i<threads.length; ++i) {
        threads[i] = new Thread(new CounterThread(counter, COUNT_TO, i%2==0));
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.run();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Serial sync - Counter value: " + counter.getValue() + " Time:
    }
}
```

INFO: Serial sync - Counter value: 0 Time: 4594 ms.

Serial Atomic Test

```
@Test
public void serialAtomicTest() throws InterruptedException {
    Counter counter = new AtomicCounter();
    Thread[] threads = new Thread[THREAD_COUNT];

    for (int i=0; i<threads.length; ++i) {
        threads[i] = new Thread(new CounterThread(counter, COUNT_TO, i%2==0));
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.run();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Serial atomic - Counter value: " + counter.getValue() + " Time
    }
}
```

INFO: Serial atomic - Counter value: 0 Time: 2422 ms.

Independent Counters Test

```
@Test
public void independentCountersTest() throws InterruptedException {
    Thread[] threads = new Thread[THREAD_COUNT];
    Counter[] counters = new Counter[threads.length];

    for (int i=0; i<threads.length; ++i) {
        counters[i] = new CounterImpl();
        threads[i] = new Thread(new CounterThread(counters[i], COUNT_TO, i%2==0
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.start();
    for (Thread thread: threads) thread.join();

    int count = 0;
    for (Counter counter: counters) count += counter.getValue();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Independent - Counter value: " + count + " Time: " + (t1-t0)
    }
}
```

INFO: Independent - Counter value: 0 Time: 93 ms.

Sync Counter Thread

```
 1 package com.tomaszmozolewski.examples.concurency.counter;
 2
 3 public class SyncCounterThread implements Runnable {
 4     Counter counter;
 5     int count;
 6     boolean isInc;
 7
 8     public SyncCounterThread(Counter counter, int count, boolean isInc) {
 9         this.counter = counter;
10         this.count = count;
11         this.isInc = isInc;
12     }
13
14     public void run() {
15         synchronized(this.counter) {
16             if (this.isInc) {
17                 while (this.count-->0) this.counter.inc();
18             } else {
19                 while (this.count-->0) this.counter.dec();
20             }
21         }
22     }
23 }
```

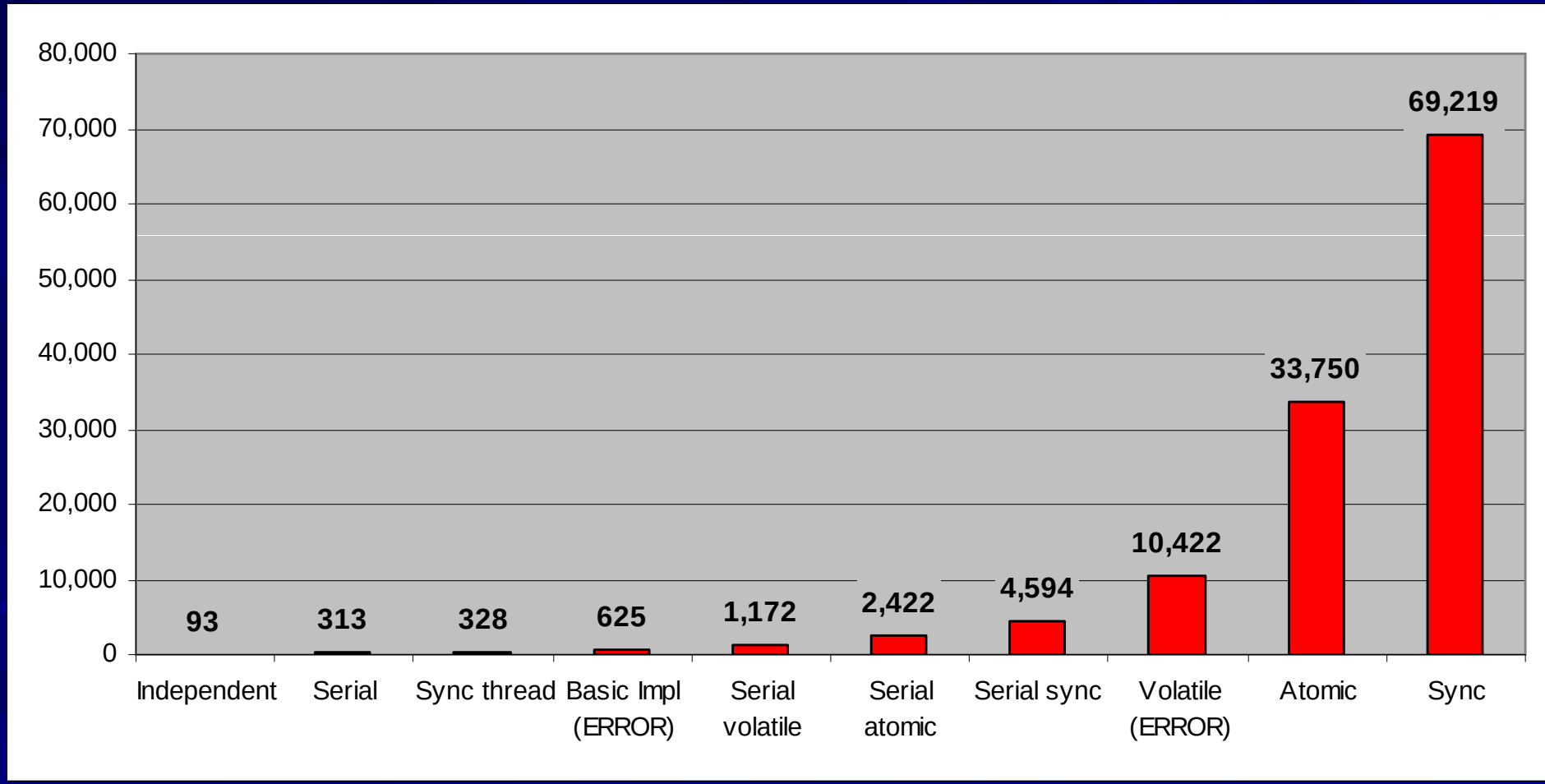
Sync Thread Test

```
@Test
public void syncCounterThreadTest() throws InterruptedException {
    Counter counter = new CounterImpl();
    Thread[] threads = new Thread[THREAD_COUNT];

    for (int i=0; i<threads.length; ++i) {
        threads[i] = new Thread(new SyncCounterThread(counter, COUNT_TO, i%2==0)
    }

    long t0 = System.currentTimeMillis();
    for (Thread thread: threads) thread.start();
    for (Thread thread: threads) thread.join();
    long t1 = System.currentTimeMillis();

    if (log.isInfoEnabled()) {
        log.info("Sync thread - Counter value: " + counter.getValue() + " Time:
    }
}
```

INFO: Sync thread - Counter value: 0 Time: 328 ms.

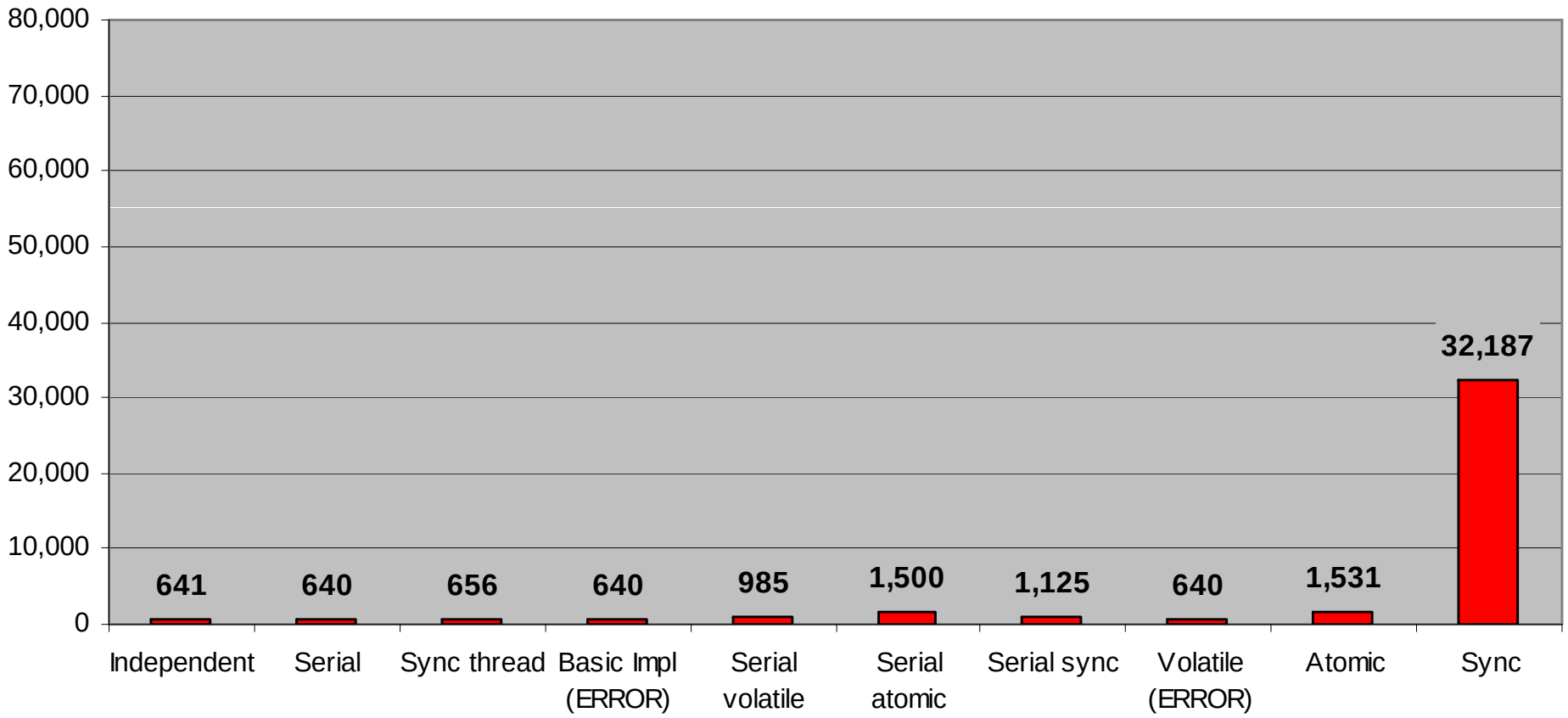
Summary (quad core CPU)

Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz



Summary (single core CPU)

Intel(R) Celeron(R) CPU @2.20GHz



Maximizing performance

- All work can be done in parallel
- Synchronization not required
- Other resources are very fast or parallel

Performance killers

- Serial work
- Interdependency
- Other resources; i.e.: reading large files
- Lock contention

Reducing contention

- Synchronize smallest section of code possible
- Synchronize on objects that need protection
- Synchronize entire loop instead of in loop synchronization
- Use independent objects whenever possible

Thread safe classes

- Performance!
- Don't mix - either make it thread safe, or not thread safe
- Document! Document! Document!
- Use existing code, don't reinvent the wheel

Common Pitfalls

- Double checked locking
- Leaking constructor
- Custom cache

Double checked locking (bad)

```
 1 package com.tomaszmozolewski.examples.concurency.doublecheckedlocking;
 2
 3 public class BadSingletonHolder implements SingletonHolder {
 4     private Singleton instance;
 5
 6     public Singleton getInstance() {
 7         if (this.instance == null) {
 8             synchronized (this) {
 9                 if (this.instance == null) {
10                     this.instance = new Singleton();
11                 }
12             }
13         }
14         return this.instance;
15     }
16 }
```

Double checked locking (good)

```
1  package com.tomaszmozolewski.examples.concurency.doublecheckedlocking;
2
3  public class GoodSingletonHolder implements SingletonHolder {
4      volatile private Singleton instance; // JDK 1.5 and higher only
5
6      public Singleton getInstance() {
7          if (this.instance == null) {
8              synchronized (this) {
9                  if (this.instance == null) {
10                     this.instance = new Singleton();
11                 }
12             }
13         }
14         return this.instance;
15     }
16 }
```

Double checked locking (better)

```
1  package com.tomaszmozolewski.examples.concurency.doublecheckedlocking;
2
3  public class BetterSingletonHolder implements SingletonHolder {
4      public Singleton getInstance() {
5          return PrivateSingletonHolder.instance;
6      }
7
8      private static class PrivateSingletonHolder {
9          public static Singleton instance = new Singleton();
10     }
11 }
```

Double checked locking (best)

```
1  package com.tomaszmozolewski.examples.concurency.doublecheckedlocking;
2
3  import org.springframework.beans.factory.BeanFactory;
4
5  public class BestSingletonHolder implements SingletonHolder {
6      private String singletonBeanName;
7      private BeanFactory factory;
8
9      public BestSingletonHolder(BeanFactory factory, String singletonBeanName)
10         super();
11         this.factory = factory;
12         this.singletonBeanName = singletonBeanName;
13     }
14
15     public Singleton getInstance() {
16         return (Singleton) factory.getBean(singletonBeanName);
17     }
18 }
```

Leaking constructor (bad)

```
1  package com.tomaszmozolewski.examples.concurency.constructorleak;
2
3  import java.util.Collection;
4
5  public class LeakingConstructor {
6      public LeakingConstructor(Collection<Object> c) {
7          super();
8          c.add(this);
9      }
10
11     public static void addNew(Collection<Object> c) {
12         new LeakingConstructor(c);
13     }
14 }
```

Leaking constructor (good)

```
1  package com.tomaszmozolewski.examples.concurency.constructorleak;
2
3  import java.util.Collection;
4
5  public class NotLeakingConstructor {
6      private NotLeakingConstructor() {
7          super();
8      }
9
10     public static NotLeakingConstructor addNew(Collection<Object> c) {
11         NotLeakingConstructor notLeakingConstructor =
12             new NotLeakingConstructor();
13
14         c.add(notLeakingConstructor);
15         return notLeakingConstructor;
16     }
17 }
```

Custom cache (bad)

```
1  package com.tomaszmozolewski.examples.concurency.cache;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  public abstract class BadCache<P, R> {
7      private Map<P, R> cache = new HashMap<P, R>();
8
9      abstract R calculate(P p);
10
11     public R get(P p) {
12         R r = null;
13
14         if (cache.containsKey(p)) {
15             r = cache.get(p);
16         } else {
17             r = calculate(p);
18             cache.put(p, r);
19         }
20         return r;
21     }
22 }
```

Custom cache (good)

```
1  package com.tomaszmozolewski.examples.concurency.cache;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  public abstract class GoodCache<P, R> {
7      private Map<P, R> cache = new HashMap<P, R>();
8
9      abstract R calculate(P p);
10
11     synchronized public R get(P p) {
12         R r = null;
13
14         if (cache.containsKey(p)) {
15             r = cache.get(p);
16         } else {
17             r = calculate(p);
18             cache.put(p, r);
19         }
20         return r;
21     }
22 }
```

```java
package com.tomaszmozolewski.examples.concurency.cache;

import java.util.HashMap;

public abstract class BetterCache<P, R> {
    private Map<P, R> cache = new HashMap<P, R>();

    abstract R calculate(P p);

    public R get(P p) {
        synchronized(cache) {
            if (cache.containsKey(p)) {
                return cache.get(p);
            }
        }

        R r = calculate(p);

        synchronized(cache) {
            if (!cache.containsKey(p)) {
                cache.put(p, r);
            }
        }

        return r;
    }
}
```

Custom cache (even better)

```
 1 package com.tomaszmozolewski.examples.concurency.cache;
 2
 3 import java.util.concurrent.ConcurrentHashMap;..
 5
 6 public abstract class EvenBetterCache<P, R> {
 7     private ConcurrentMap<P, R> cache = new ConcurrentHashMap<P, R>();
 8
 9     abstract R calculate(P p);
10
11     public R get(P p) {
12         R r = null;
13
14         if (cache.containsKey(p)) {
15             r = cache.get(p);
16         } else {
17             r = calculate(p);
18             cache.putIfAbsent(p, r);
19         }
20         return r;
21     }
22 }
```

Reference

- Java Concurrency in Practice by Brian Goetz et al.
  http://www.amazon.com/Java-Concurrency-Practice-Brian-Goetz/dp/product-description/0321349601
- Sun Java Concurrency Tutorial
  http://java.sun.com/docs/books/tutorial/essential/concurrency/
- Java 1.5 concurrent packages
  http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/util/concurrent/package-summary.html
  http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/util/concurrent/atomic/package-summary.html
  http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/util/concurrent/locks/package-summary.html
- Java 1.5 ProcessBuilder
  http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/lang/ProcessBuilder.html
- Using ThreadLocal to implement a per-thread Singleton
  http://www.ibm.com/developerworks/java/library/j-threads3.html
- Double checked locking
  http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html
- Reducing contention
  http://www.ibm.com/developerworks/java/library/j-threads2.html