Parallel programming in Java

Sergey Salishev, SPbSU and Intel Labs, Russia
Outline

- Why is parallel programming so hard?
- Kinds of parallelism
- Implicit and explicit parallelism
- Most popular parallel model either
- Parallelism in the Java language
- java.util.concurrent
Why is parallel programming so hard?

- Human consciousness is sequential
- It is hard not only in programming
  - Doing two things simultaneously
  - Leading a team of co-workers
- Parallel programming is not just coding
  - In nontrivial cases it is the essence of program architecture
- To write complex parallel program programmer needs to think as multiple parallel entities
  - It can develop into a split personality
Kinds of parallelism

- Like sequential but faster
  - Do you need an explicit parallelism?
  - Sequential equivalence saves the day

- You cannot do without (aka Concurrency)
  - Be natural, OOP was invented for it
  - Smart Little Creature modeling (TRIZ)
    - Focused on single task at a time
    - Specialized
    - Minimal number
  - Add sequential arbiters (trafficlights)
Implicit parallelism

- Data parallelism
  - Chunks of data are processed independently by the same program

- Pipeline
  - One chunk of data is processed by multiple programs in same order

- Task parallelism
  - Independent branches in data flow are processed in parallel

- Transactional parallelism
  - Independent branches in control flow executed in parallel
  - Atomicity, Consistency, Isolation, Durability (ACID)
  - Like task parallelism but we don’t know data dependency beforehand
Explicit parallelism

- Passive objects – just data with operations
- Active objects (Actors) – act like real persons
  - Communicate by creating and processing events (messages)
  - Synchronous communication
    - Hoare’s monitors, synchronous objects
    - Rendezvous (wait/notify), \( P1 \{e\} P2 \), 0-queue
- Asynchronous communication
  - Unbounded message queues
  - Joins, predicates on multiple events
    - Join-calculus, Join Java, HW Join Java, JOCaml, C++ Boost.Join, C\( \omega \)
Most popular parallel model either

- If your system communicates with one external object or messages are independent from each other
  - No explicit parallelism is needed
- In case of multiple objects with communication sessions
  - Active Proxy object in the system for each
  - All shared data in DB (SQL, in-memory, NoSQL, whatever)
  - Synchronize data in DB using transactions
- Oops. Is it like a web service? Yes it is.
  - It is most popular parallel model in the world and in Java
  - It is easy, stable, scalable
  - Every web programmer can do it right
Parallel facilities in Java

- **Super High level (external libs)**
  - Application server, Big Data platform: Tomcat, Glass Fish, Resin, Hadoop
  - Database: Java DB, H2, HSQLDB, HBase

- **High level (java.util.concurrent)**
  - ForkJoinTask (Java 7), Future, Task, Executor, ThreadPool
  - Copy on write collections
  - Concurrent collections

- **Medium level (java.util.concurrent + java.lang.Thread)**
  - Thread, BlockingQueue

- **Low level (java.util.concurrent + java.lang.Thread + core language)**
  - Thread, Monitor
  - Atomics
  - Synchronizers (Barriers, Semaphore)
  - Locks
  - park/unpark
Problem with threads

- Basic Thread programming model
  - Shared Memory, Rendezvous
  - Almost naive implementation of minimal models of parallelism (PRAM + CSP)

- Threads are like a parallel assembly language
  - You only really need them when developing frameworks
  - Parallel Random-Access Machine (PRAM)
    - Mathematical model of shared memory
    - Accesses are atomic, can be used for busy loop sync
  - Communicating Sequential Processes (CSP)
    - Term rewriting parallel formalization based on rendezvous
    - Rendezvous is \((P1|\{e}\|P2), 0\)-queue
Parallelism in Java language

- **Threads**
  ```java
  new Thread(new Runnable() {
    public void run() {
      ...
    }
  }).start();
  ```

- **Synchronized, maps to Monitor lock/unlock**
  ```java
  synchronized void meth() {
    synchronized(other) {
      ...
    }
  }
  ```

- **Wait/notify**
  ```java
  synchronized(other) {
    other.notifyAll();
    other.wait();
  }
  ```

- **volatile fields**

- **Causal memory model (JLS 3.0, 17.4)**
Causal memory model in Java

- Memory writes can be physically reordered by
  - Simultaneous execution on different processors
  - Superscalar processor
  - Memory caches
  - Optimizing compiler
- Only observable event order is important (Lamport’s logical clock)
- Memory ops on one thread are observed in execution order on that thread
- Memory ops are atomic (except for long/double for embedded VM)
  UNDEFINED OBSERVABLE ORDER from other threads
- Before Java 1.5 only synchronized operations were ordered
  - Expected memory order implicitly resulted from cache coherence in CPU
- Causal memory model (JLS 3.0, 17.4) introduced in Java 1.5
  - Memory operations are observed in execution order by a synchronized observer
  - Specification was changed to accommodate NUMA memory model introduced in AMD Opteron used in Sun Enterprise Servers
Cache coherency and NUMA

- Cache coherency provides consistency of data stored in local caches of a shared resource.
- Older multiprocessor systems were symmetric (SMP) with single shared memory.
- Older coherency protocols provided sequential consistency.
- Multiprocessors with distributed memory provide weak consistency only to reduce performance overhead.
  - Consistency is guaranteed only at accesses to synchronization variables.
Synchronized operations

- *Volatile read/write*
- *Monitor lock/unlock*
- Synthetic first and last thread operations
- Thread start and termination
- Thread.interrupt() and it’s detection
- There is a total order of synchronized operations
  - It is consistent with execution order in each thread
**Synchronizes-with**

- M.unlock with following M.lock
- V.write following V.read
- Thread.start() with first thread operation
- Default value init (0) with first operation of every thread
- Last thread operation with thread termination detection (Thread.isAlive(), Thread.Join())
- Thread.interrupt() with interrupt detection (InterruptedException, Thread.interrupted(), Thread.isInterrupted())
Happens-before

- Let $x, y$ be operations. $hb(x, y)$ means that $x$ happens before $y$.
- $hb$ is a partial order of operations
- $hb(x, y)$ if
  - $x, y$ are executed on the same thread and $x$ is executed before $y$
  - $x$ is the last operation of object constructor, and $y$ is the first operation of `Object.finalize()` of the same object
  - $x$ synchronizes-with $y$
  - $hb(x, z)$ and $hb(z, y)$ – transitivity
  - $x$ (write) precedes $y$ (read) by `final` field semantics (JLS 3.0, 17.5)
- **ATTENTION!** If $z$ is unordered relative to $x$ and $y$, then $z$ doesn’t know the order of $x$ and $y
Examples of operation order

synchronized void m() {
    notifyAll();
    wait();
}

- notifyAll() does not awake the following wait

class A {
    final B b;
    A() {
        b = new B();
    }
}

- b contains the reference to fully initialized object
Example (lazy init)

**Incorrect**
```java
class A {
    private R r;
    R getR() {
        R result = r;
        if (result == null) {
            synchronized {
                if (r == null) {
                    result = r = new R();
                }
            }
        }
        return result;
    }
}
```

**Correct**
```java
class A {
    private volatile R r;
    R getR() {
        R result = r;
        if (result == null) {
            synchronized {
                if (r == null) {
                    result = r = new R();
                }
            }
        }
        return result;
    }
}
```

Better version for singleton (works due to lazy class loading required by JVM spec.)
```java
class A {
    static R getR() { return RHolder.INSTANCE; }

    static class RHolder {
        static final R INSTANCE = new R();
    }
```
java.util.concurrent

- ForkJoinTask, Future, Task, Executor, ThreadPool

class Fibonacci extends RecursiveTask<Integer> {
    final int n;
    Fibonacci(int n) { this.n = n; }
    public Integer compute() {
        if (n <= 1) { return n; }
        ForkJoinTask<Integer> f1 = new Fibonacci(n - 1).fork();
        ForkJoinTask<Integer> f2 = new Fibonacci(n - 2);
        return f2.invoke() + f1.join();
    }
}

public class JUCTest {
    static final ForkJoinPool mainPool = new ForkJoinPool();

    public static void main(String[] args) {
        int res = mainPool.invoke(new Fibonacci(10));
        System.out.println(res);
    }
}
ForkJoin and memory consistency Example

class SortTask extends RecursiveAction {
  final long[] array;
  final int lo;
  final int hi;
  SortTask(long[] array, int lo, int hi) {...}
  protected void compute() {
    if (hi - lo < THRESHOLD)
      sequentiallySort(array, lo, hi);
    else {
      int mid = (lo + hi) >>> 1;
      invokeAll(new SortTask(array, lo, mid),
                 new SortTask(array, mid, hi));
      merge(array, lo, hi);
    }
  }
}
java.util.concurrent

- Dynamic load balancing (sum)

class Sum extends RecursiveTask<Double> {
    final double[] array; final int lo, hi;
    Sum next; // keeps track of right-hand-side tasks
    Sum(double[] array, int lo, int hi, Sum next) {...}
    double sumAtLeaf(int l, int h) {...}
    protected Double compute() {
        int l = lo; int h = hi; Sum right = null;
        while (h - l > 1 && getSurplusQueuedTaskCount() <= 3) {
            int mid = (l + h) >>> 1;
            right = new Sum(array, mid, h, right);
            right.fork();
            h = mid;
        }
        double sum = sumAtLeaf(l, h);
        while (right != null) {
            if (right.tryUnfork()) { // directly calculate if not stolen
                sum += right.sumAtLeaf(right.lo, right.hi);
            } else { sum += right.join(); }
            right = right.next;
        }
        return sum;
    }
}
java.util.concurrent

• Copy on write collections
  • CopyOnWriteArrayList<E>
    A thread-safe variant of ArrayList in which all mutative operations (add, set) are implemented by making a fresh copy of the underlying array
  • CopyOnWriteArraySet<E>
    A Set that uses an internal CopyOnWriteArrayList for all of its operations

• Concurrent collections
  • ConcurrentHashMap<K,V>
    A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates
  • ConcurrentLinkedDeque<E>
    An unbounded concurrent deque based on linked nodes
  • ConcurrentLinkedQueue<E>
    An unbounded thread-safe queue based on linked nodes
  • ConcurrentSkipListMap<K,V>
    A scalable concurrent ConcurrentNavigableMap implementation
  • ConcurrentSkipListSet<E>
    A scalable concurrent NavigableSet implementation based on a ConcurrentSkipListMap.
java.util.concurrent

- BlockingQueue
  - ArrayBlockingQueue< E >
    A bounded blocking queue backed by an array
  - DelayQueue< E extends Delayed >
    An unbounded blocking queue of Delayed elements, in which an element can only be taken when its delay has expired
  - LinkedBlockingDeque< E >
    An optionally-bounded blocking deque based on linked nodes
  - LinkedBlockingQueue< E >
    An optionally-bounded blocking queue based on linked nodes
  - LinkedTransferQueue< E >
    An unbounded TransferQueue based on linked nodes
  - PriorityBlockingQueue< E >
    An unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations
  - SynchronousQueue< E >
    A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa
Atomics
- AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference\<V> Atomic scalar variable wrapper
- AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray\<E> Array with atomic entries
- AtomicMarkableReference\<V> Atomic reference+boolean
- AtomicStampedReference\<V> Atomic reference+int

Example
```java
class Sequencer {
    private final AtomicLong sequenceNumber = new AtomicLong(0);
    public long next() {
        return sequenceNumber.getAndIncrement();
    }
}
```
java.util.concurrent

• Synchronizers
  • **Semaphore** is a classic concurrency tool.
  • **CountDownLatch** is a common utility for blocking until a given number of signals, events, or conditions hold
  • **CyclicBarrier** is a resettable multiway synchronization point useful in some styles of parallel programming
  • **Phaser** provides a more flexible form of barrier that may be used to control phased computation among multiple threads
  • **Exchanger** allows two threads to exchange objects at a rendezvous point, and is useful in several pipeline designs.
java.util.concurrent

• Locks
  • ReentrantLock A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities
  • ReentrantReadWriteLock An implementation of ReadWriteLock supporting similar semantics to ReentrantLock
java.util.concurrent

- `LockSupport.park()` Disables the current thread for thread scheduling purposes unless the permit is available
- `LockSupport.unpark()` Makes available the permit for the given thread, if it was not already available
- Along with Atomics it is the foundation and only non-Java part of java.util.concurrent
- Can be used to build your own types of locks
java.util.concurrent

- Example of custom lock

```java
class FIFOMutex {
    private final AtomicBoolean locked = new AtomicBoolean(false);
    private final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();
    public void lock() {
        Thread current = Thread.currentThread();
        waiters.add(current);
        // Block while not first in queue or cannot acquire lock
        while (waiters.peek() != current || !locked.compareAndSet(false, true)) {
            LockSupport.park(this);
        }
        waiters.remove();
    }
    public void unlock() {
        locked.set(false);
        LockSupport.unpark(waiters.peek());
    }
}
```