Multi-core in JVM/Java

- Concurrent programming in java
- Prior Java 5
- Java 5 (2006)
- Java 7 (2010)
- Other topics

Basic concurrency in Java

- Java Memory Model describes how threads interact through memory
  - Single thread execution within thread as-if-serial
  - Partial order in communication between thread
- Basic concurrency construct included in language
  - Threads
  - Synchronization

Processes and Threads

- Basically, a Java virtual machine run as a single process
- Programmer can implement concurrency by using multiple threads

- It is also create a new process by instantiating ProcessBuilder object, e.g

```
ProcessBuilder pb = new ProcessBuilder("command", "arg1”, "arg2");
Process p = pb.start();
```

Java Threads

- Provides similar features than Posix threads
- Java thread is an instance of Thread class
- Commonly used methods:

```
public void run()
public synchronized void start()
public final synchronized void join(long milliseconds)
public static void yield()
public final int getPriority()
public final void setPriority(int newPriority)
```

Java Threads

- Can be used either by subclassing *Thread* or implementing *Runnable* interface.

```
public class MyThread1 extends Thread {
    public void run() {
        //thread code
    }
    public static void main(String args[]) {
        (new MyThread1()).start();
    }
}

public class MyThread2 implements Runnable {
    public void run() {
            // thread code
    }
    public static void main(String args[]) {
        (new Thread(new MyThread2())).start();
    }
}
```

Synchronized Methods

- Only one thread can execute objects synchronized method(s) at time
- e.g:

```
Public class SynchronizedCounter {
    public synchronized void update(int x) {
            count += x;
    }
    public synchronized void reset {
            count = 0;
    }
}
```

Synchronized Statements

- Finer-grained synchronization
- Specify the object which provides a lock

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) { c1++; }
    }
    public void inc2() {
        synchronized(lock2) { c2++;  }
    }
}
```

Java 5 (2006)

- **java.util.concurrent**
  - Utility classes commonly useful in concurrent programming (e.g. executors, thread pools, concurrent containers)
- **java.util.concurrent.atomic**
  - A small toolkit of classes that support lock-free thread-safe programming on single variables
- **java.util.concurrent.locks**
  - Interfaces and classes for locking and waiting for conditions

Atomic Objects

- Package java.util.concurrent.atomic supports lock-free atomic operations on single variables e.g:

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

- Example of methods (AtomicInteger)

```
int addAndGet(int delta);
boolean compareAndSet(int expect, int update);
int decrementAndGet();
int incrementAndGet();
```

Lock objects

- Package java.util.concurrent.locks provides interfaces and classes for locking and waiting for conditions
- Allow more flexibility for using locks
- Interfaces:

  ReadWriteLock

  Condition

  Lock

Lock objects, example

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException
   {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
```

Executor framework

- Allows to create custom thread management for *Runnable* tasks
- Decouples task submission from the mechanics of how each task will be run
- Some interfaces:
  - *Callable*
  - *Future*
  - *Executor*
  - *ExecutorService*
  - *ScheduledExecutorService*

Executor

- Examples:

```
class DirectExecutor implements Executor
{
    public void execute(Runnable r) {
        r.run();
    }
}
```

```
class ThreadPerTaskExecutor implements Executor
{
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}
```

Executor

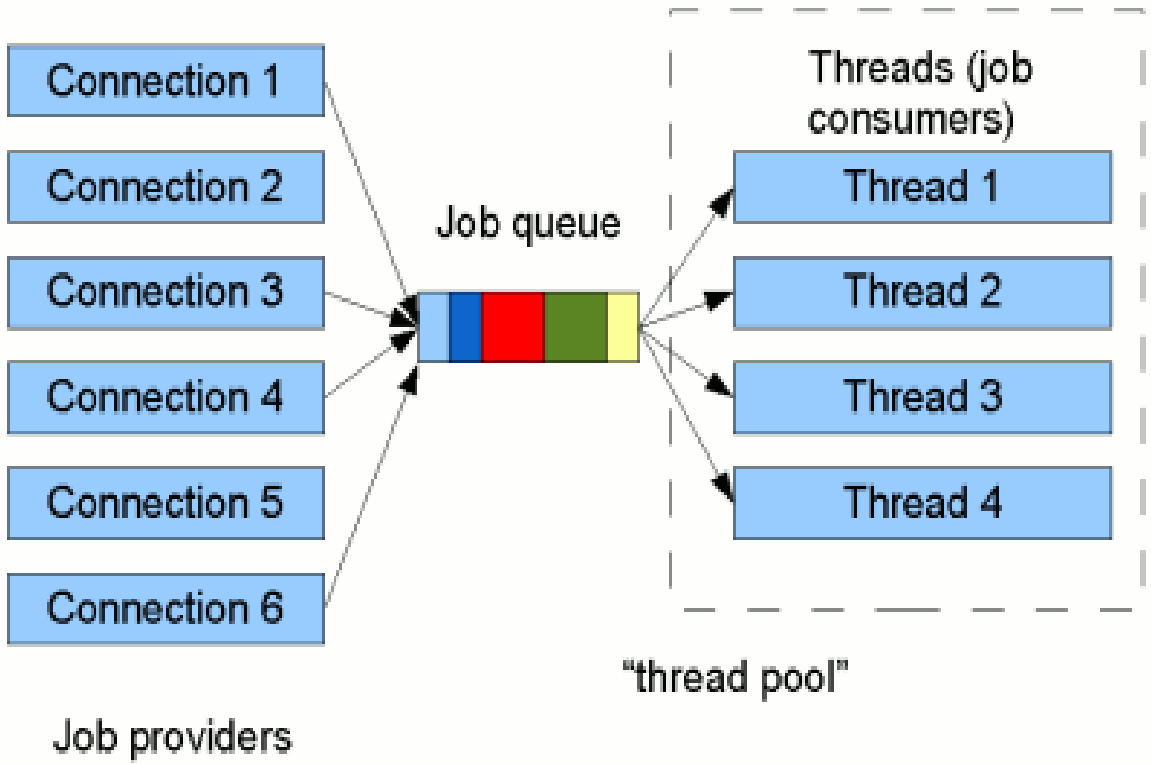
- Example of ScheduledExecutorService

```
class BeeperControl {
   private final ScheduledExecutorService scheduler =
      Executors.newScheduledThreadPool(1);

   public void beepForAnHour() {
       final Runnable beeper = new Runnable() {
              public void run() { System.out.println("beep"); }
          };
       final ScheduledFuture<?> beeperHandle =
          scheduler.scheduleAtFixedRate(beeper, 10, 10, SECONDS);
          scheduler.schedule(new Runnable() {
              public void run() { beeperHandle.cancel(true); }
          }, 60 * 60, SECONDS);
   }
}
```

ThreadPoolExecutor

- Reuse threads for multiple tasks



Queues

- *ConcurrentLinkedQueue* class defines an unbounded non-blocking thread-safe FIFO
- *BlockingQueue* interface defines a thread-safe blocking queue
  - Classes: *LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue, DelayQueue*
- *BlockingDequeue* interfaces defines a thread-safe double ended queue

Synchronizers

- *Semaphore*
- *CountDownLatch*
- *CyclicBarrier*
- *Exchanger*

Concurrent Collections

- *ConcurrentHashMap*
- *CopyOnWriteArrayList*
- *CopyOnWriteArraySet*

Concurrency in Java 7

- Target release data early 2010
- Number of cores increases -> need for more and finer grained parallelism to keep processor cores busy

- Fork-join framework
- ParallelArray

Fork-join framework

- Divide and conquer approach

```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```

Fork-join framework

- java.util.concurrent.forkjoin
- Desinged to minimize per-task overhead
- *ForkJoinTask* is a lightweight thread
- *ForkJoinPool* hosts *ForkJoinExecutor*
- Work Stealing

Example

```
class MaxSolver extends RecursiveAction {
        private final MaxProblem problem;
        int result;

        protected void compute() {
                if (problem.size < THRESHOLD)
                        result = problem.solveSequentially();
                else {
                        int m = problem.size / 2;
                        MaxSolver left, right;
                        left = new MaxSolver(problem.subproblem(0, m));
                        right = new MaxSolver(problem.subproblem(m,problem.size));
                        forkJoin(left, right);
                        result = Math.max(left.result, right.result);
                }
        }
}

ForkJoinExecutor pool = new ForkJoinPool(nThreads);
MaxSolver solver = new MaxSolver(problem);
pool.invoke(solver);
```

Performance

Results of Running select-max on 500k-element Arrays on various systems

Threshold=	500k	50k	5k	500	50
Pentium-4 HT (2 threads)	1.0	1.07	1.02	0.82	0.2
Dual-Xeon HT (4 threads)	0.88	3.02	3.2	2.22	0.43
8-way Opteron (8 threads)	1.0	5.29	5.73	4.53	2.03
8-core Niagara (32 threads)	0.98	10.46	17.21	15.34	6.49

- Significant performance improvement can be gained if sequential threshold is reasonable
- Portable performance

ParallelArray

- Specify aggregate operation on arrays at higher abstraction layer
- Parallel Array framework automates fork-join decomposition for operation on arrays
- Supported operations:
  - Filtering
  - Mapping
  - Replacement
  - Aggregation
  - Application

ParallelArray Example

```
ParallelArray<Student> students = new ParallelArray<Student>(fjPool, data);
double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .max();

public class Student {
    String name;
    int graduationYear;
    double gpa;
}

static final Ops.Predicate<Student> isSenior = new Ops.Predicate<Student>() {
    public boolean op(Student s) {
        return s.graduationYear == Student.THIS_YEAR;
    }
};

static final Ops.ObjectToDouble<Student> selectGpa = new Ops.ObjectToDouble<Student>()
{
    public double op(Student student) {
        return student.gpa;
    }
};
```

Parallel Array Performance

Table 1. Performance measurement for the max-GPA query
(Core 2 Quad system running Windows)

Threads	**1**	**2**	**4**	**8**
Students				
1000	1.00	0.30	0.35	1.20
10000	2.11	2.31	1.02	1.62
100000	9.99	5.28	3.63	5.53
1000000	39.34	24.67	20.94	35.11
10000000	340.25	180.28	160.21	190.41

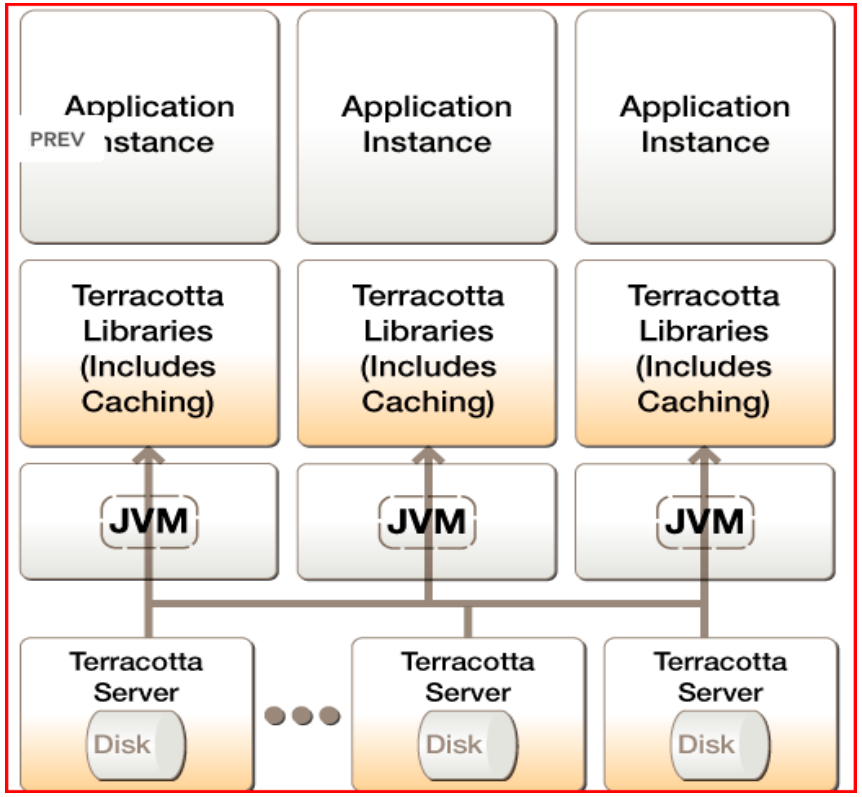
- Best speedup (2x-3x) achieved when nof cores equals to nof threads (as expected)

Other topics

- Cluster computing (Terracotta)
- Stream Programming (Pervasive DataRush)
- Highly scalable lib
- Transactional Memory

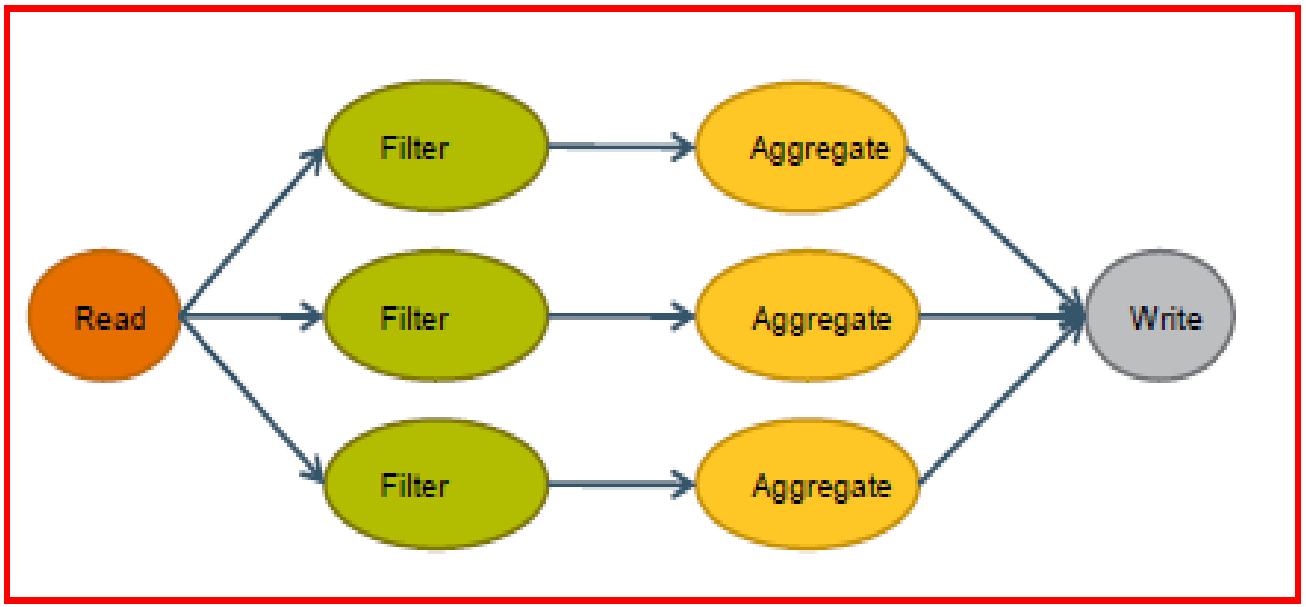
Terracotta

- Open source infrastructure to scale Java application to many computers
  - http://www.terracotta.org
- Transparent to programmer
  - Converts multi-threaded application to a multi-JVM (clustered) application.
  - Specify objects need to be shared across cluster



Stream Programming

- http://www.pervasivedatarush.com/
- Based on dataflow graph, computation nodes interconnected by queues



Highly scalable Lib

- Concurrent and Highly Scalable Collection
- http://sourceforge.net/projects/high-scale-lib
- Replacements for the java.util.* or java.util.concurrent.* collections

Highly Scalable Lib

- ConcurrentAutoTable
  - auto-resizing table of longs, supporting low-contention CAS operations
- NonBlockingHashMap
  - A lock-free implementation of ConcurrentHashMap
- NonBlockingSetInt
  - A lock-free bit vector set

- Liner scaling (tested up to 768 CPUs)

Transactional memory and Java

- Sequence of memory operations that execute completely (**commit**) or have no effect (**abort**)

```
atomic {
        if (inactive.remove(p))
        active.add(p);
}
```

- STM or HTM
- Still a research subject

Transactional memory

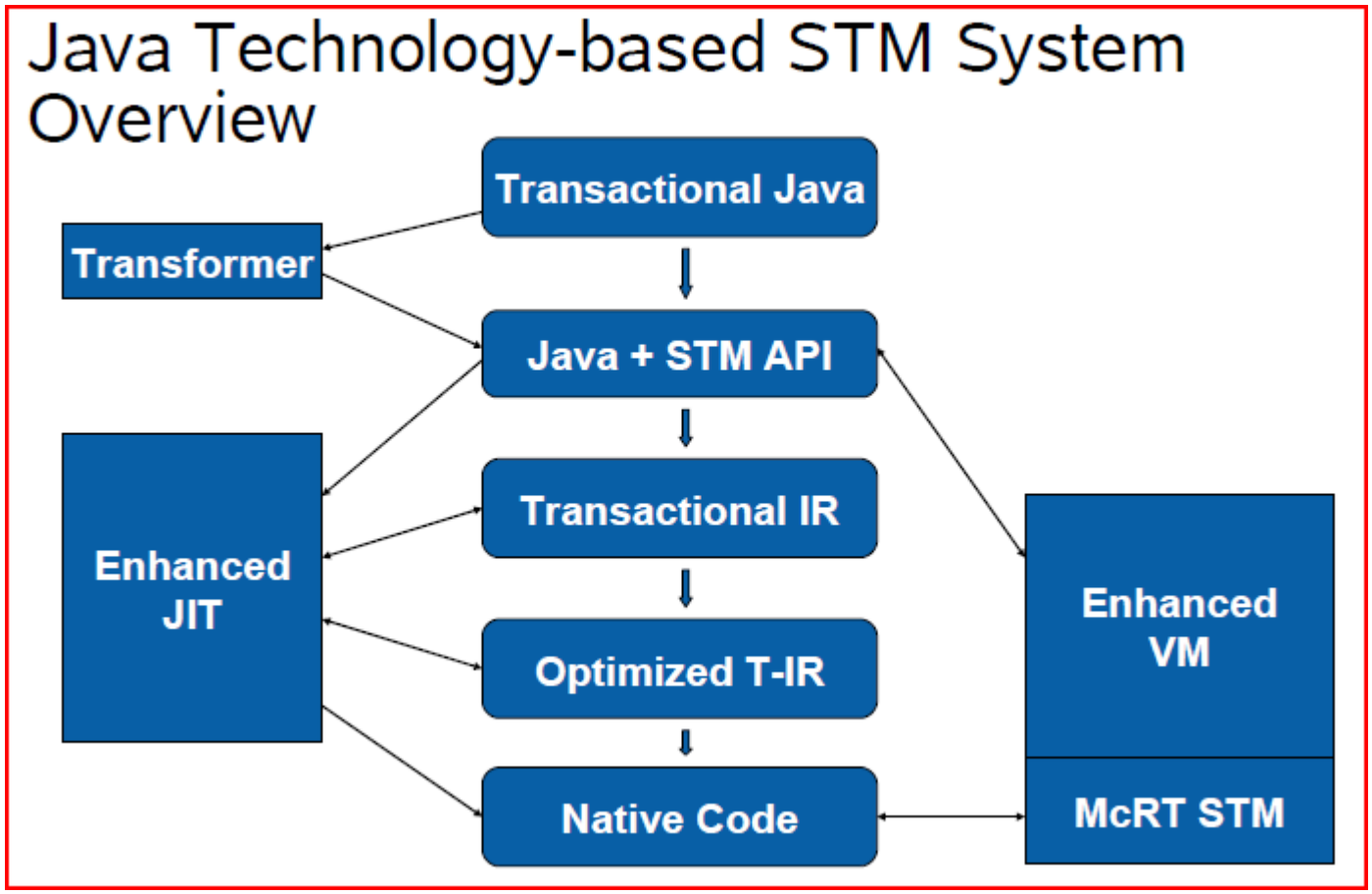
Pros

- Transactions compose
- Can't acquire wrong lock
- No deadlocks
- No Priority inversion

Cons/problems

- How to roll-back I/O?
- Live-lock
- Mixing of transactional and non-transactional code
- Performance

Example (McRT STM)



http://developers.sun.com/learning/javaoneonline/2008/pdf/TS-6316.pdf?cid=925329