Scaling Java Applications Through Concurrency

Sharing Resources Among Parallel Workers

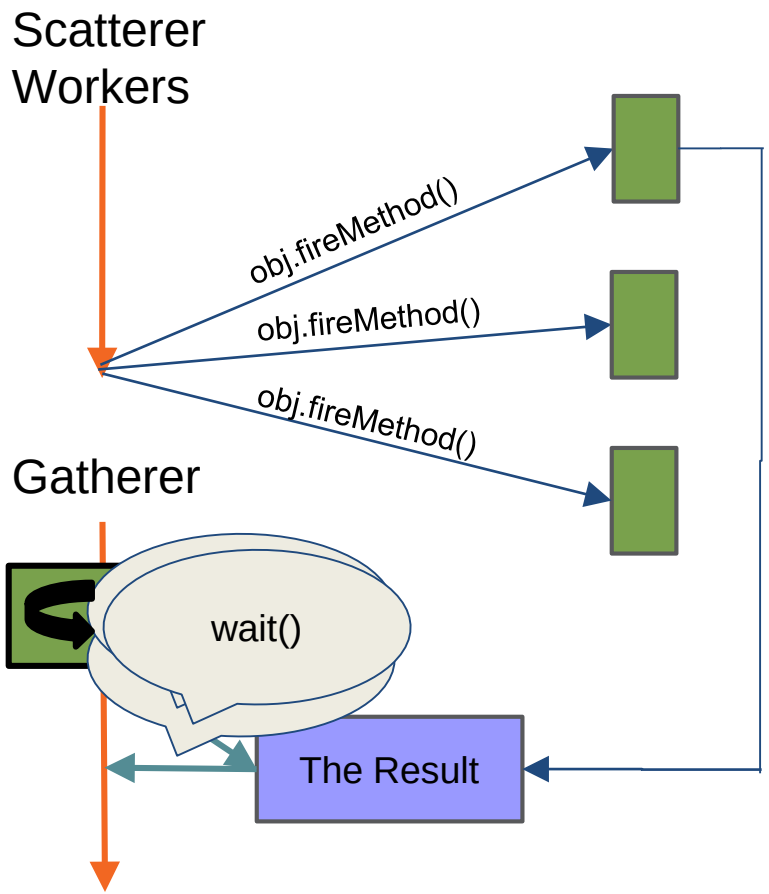
Josh Cummings

@jzheaux | tech.joshcummings.com

Identity Pipeline: Parallel Workers

- **Reads in a stream of identity-related data**
- Normalizes and verifies the contents of each identity
- Merges and persists the identities in an in-memory cache
- Records aggregate identity statistics
- Notifies an error queue if any of the above fails

Scatter-Gather Pattern



- Ask several independent workers their vote or opinion on the same question, short-circuiting when a quorum is reached
- Segment work across several workers
- Ask several workers to perform the same operation for reliability or performance guarantee
- Workers operate in parallel for greater scalability

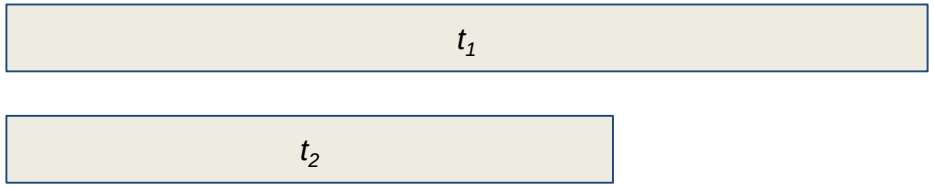
Future

- An Object that represents the result that a Worker will eventually provide
- Allows the submitter to block on Workers' results
- Futures can be returned to layers that are prepared to block on the response

```
// deploy a worker thread, storing a Future
// for later blocking

Future<Identity> identityHolder = pool.submit(() -> reader.read(source));

Identity identity = identityHolder.get(); // blocks until thread completes

List<Future<Identity>> identitiesHolder = pool.invokeAll(listOfCallables );
```

ExecutorCompletionService

- Adds completed tasks to a queue from which results can be retrieved in the order that they were completed
- Reduces real time execution



$t = t_1 + p_1 + p_2$ *(block on first)*

$t = t_2 + p_2 + (t_1 - t_2 - p_2) + p_1 = t_1 + p_1$ *(block on shortest)*

```
ExecutorCompletionService ecs = new
ExecutorCompletionService(pool);

// deploy several worker threads
for ( Task t : tasks ) {
    ecs.submit(() -> reader.read(source));
}

Identity identity = ecs.take().get(); //
blocks until first thread completes
```

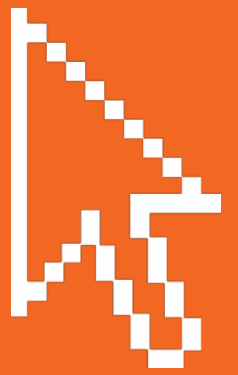
Identity Pipeline: Parallel Workers

- Reads in a stream of identity-related data
- Normalizes and verifies the contents of each identity
- Merges and persists the identities in an in-memory cache
- **Records aggregate identity statistics**
- Notifies an error queue if any of the above fails

Thread-safe Containers

NOT Thread Safe	Thread Safe
ArrayList	ConcurrentLinkedQueue, BlockingQueue
HashMap	ConcurrentHashMap
incrementation	AtomicInteger, LongAdder

Compound statements are not atomic ⇒ They are not naturally thread-safe



ReentrantLock

- Abstracts the monitor into an object, allowing acquisition and release patterns other than simple code blocks
- Allows for multiple Conditions to be created from a single lock for more sophisticated waiting patterns.
- Maintains the same atomicity and visibility semantics as **synchronized**
- May be faster than the **synchronized** keyword.

```
Lock lock = new ReentrantLock();
Condition circumstance =
lock.newCondition();

// wrapping an unsafe invocation in a lock
// identical to wrapping in synchronized
public void guardedInvocation() {
        lock.lock();
        try {

        obj.threadUnsafeInvocation();
        } finally {
                lock.unlock();
        }
}
```

Identity Pipeline: Parallel Workers

- Reads in a stream of identity-related data
- Normalizes and verifies the contents of each identity
- **Merges and persists the identities in an in-memory cache**
- Records aggregate identity statistics
- Notifies an error queue if any of the above fails

ReentrantLock#tryLock

- Requests lock, returning immediately if the lock is already held
- Usage may allow thread to “barge” past other threads currently waiting on lock() to return

```
ReentrantLock lock = new ReentrantLock();

// the underlying invocation will only occur
// if both locks can be acquired
public void complexInvocation() {
        if ( lock1.tryLock() && lock2.tryLock()
) {
                try {

        obj.threadUnsafeInvocation();
                } finally {
                        lock1.unlock();
                        lock2.unlock();
                }
        }
}
```

Review

- Scatter-Gather is a pattern for splitting work among several workers
- **ExecutorCompletionService** and **Future** can facilitate promise architectures
- Java **Concurrentxxx**, **Atomicxxx**, and **ReentrantLock** offer richer semantics than standard concurrency primitives