

Concurrency and Parallelism in Java 7

Josua Krause


Department of Computer and Information Science
University of Konstanz, Konstanz, Germany

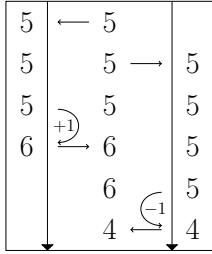
September 30, 2012

1 Introduction

The package `java.util.concurrent` provides classes to easily deal with *concurrency* and *parallelism* in Java since version 5. A common definition of concurrency is multiple tasks executing in overlapping time spans whereas parallelism is the simultaneous execution of multiple threads [Sta09]. Parallelism can be further divided into *data parallelism* and *task parallelism*. Data parallelism is the parallel execution of tasks with read-only shared data. Task parallelism allows concurrent access to shared data (e.g. *consumer-producer* tasks). To prevent multiple tasks from modifying data at the same time *locking* methods are required.

An important feature of the package `java.util.concurrent` is that it preserves *happens-before* relationships [Ora11b], i.e. that any action in a thread *happens-before* every other action which is defined later in the program code. This seems to be an obvious property but considering an optimizing compiler which can alter the execution order of actions, the compiler could start a thread immediately on its creation rather than the call of the method `start()`. Although improving the execution speed of an application, such an optimization would be counter-intuitive and could lead to errors that are hard to locate. Two goals of the package are granular locking and reuse of threads. Naïve implementations of the provided functionalities either do not reach those goals or are likely to be error-prone.

In this report we will first have a brief look at concurrent data structures together with atomic variables and thread locals. We will then discuss thread pools and look closely at the new fork-join pool introduced in Java 7 [Ora11a]. In the end we will have a short outlook on concurrency and parallelism in future Java versions. Throughout this report classes that are newly introduced in Java 7 are pointed out and specially marked  Java 7 like on this sentence.



(a) Lost update problem.

```

1 public int incrementAndGet() {
2     for (;;) {
3         int current = get();
4         int next = current + 1;
5         if (compareAndSet(current, next))
6             return next;
7     }
8 }

```

(b) Increment method of `AtomicInteger` [Lea08].

Figure 1: The lost update problem (a) and a solution from Java (b).

2 Concurrent Data Storage

Concurrent execution can cause inconsistencies if two tasks access the same data. To prevent this, locking can be used. The `Collections` framework provides static `synchronizedX(...)` wrappers to convert any collection into a thread-safe version. However, additional synchronization is needed when calling a sequence of methods. This approach locks the whole data structure letting all threads, that request access to it, wait until the current operation is completed.

Concurrent collections in Java do not need explicit synchronization but a user has to be aware of the fact that the collection may change externally. Consequently, such collections never throw a `ConcurrentModificationException`. Examples of concurrent collection are `ConcurrentHashMap`, `ConcurrentLinkedList`, `ConcurrentSkipListMap`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, etc..

For consumer-producer tasks the concurrency framework provides blocking queues (`LinkedBlockingQueue`, `PriorityBlockingQueue`, etc.) that let the consuming threads sleep until the queue is filled. Producing threads sleep when the queue reaches its maximal size until more elements can be inserted. Besides those tasks the best practice is to plan dependencies to avoid the need of concurrent data structures at all.

3 Atomic Variables

Atomic variables can be modified by one thread at a time without the need of synchronization. Therefore they are a solution to the *lost update problem*. This problem occurs when two threads attempt to modify a resource simultaneously while reading and writing are considered distinct operations (Figure 1a).

Atomic variables are implemented in Java using non-cached `volatile` variables and repeated *compare and set* operations. With `compareAndSet(...)` a variable is only updated if the current value equals the expected value, i.e. when the variable is not externally updated during the computation of the next value. If `compareAndSet(...)` fails the variable is freshly read again and the calculation is repeated (Figure 1b). Therefore, a thread may *starve* due to repeated concurrent modification.

The method `compareAndSet(...)` can be replaced with a single assembler instruction via *compiler intrinsics* [Bar12] (much like `Integer.bitCount(int)` is substituted with the `POPCNT` assembler instruction on some systems). Atomic variables exist for every primitive type.

4 Thread Locals

Thread locals are objects that are unique per thread. Every thread has its own instance of a given thread local class. This is useful when multiple threads call the same method simultaneously. Also it removes the necessity to pass intermediate results as arguments of methods in a long running task.

The class `ThreadLocalRandom` for example provides access to guaranteed independent random numbers that do not need locking. Without locking, a shared `Random` instance may yield same results for different threads when the calls to e.g. `nextInt()` overlap. Synchronizing the shared instance reduces parallelism and is therefore not desirable (Figure 2a). Furthermore creating `Random` instances within the method produces a lot of garbage and is also implicitly synchronized to ensure distinctive seeds for each constructor call.

 Java 7

A solution to those problems offer thread local objects. An instance of `ThreadLocalRandom` (Figure 2b) is created once per thread and stored directly in the `Thread` object. Therefore after the creation of the instance no further synchronization is needed anymore to guarantee independent random numbers. Although special care has to be taken for own implementations of `ThreadLocal` to keep the memory usage low. Objects must be explicitly removed from a thread to reclaim memory which can otherwise lead to memory leaks.

```

1  private Random rnd = new Random();
2
3  // method that is called in
4  // parallel by multiple threads
5  public void foo() {
6      // ...
7      int r;
8      synchronized (rnd) {
9          r = rnd.nextInt();
10     }
11     // ...
12 }
```

(a) Shared `Random` instance.

```

1  // method that is called in
2  // parallel by multiple threads
3  public void foo() {
4      // ...
5      ThreadLocalRandom rnd =
6          ThreadLocalRandom.current();
7      int r = rnd.nextInt();
8      // ...
9  }
```

(b) `ThreadLocalRandom`.

Figure 2: Locking a shared `Random` instance vs using a `ThreadLocalRandom`.

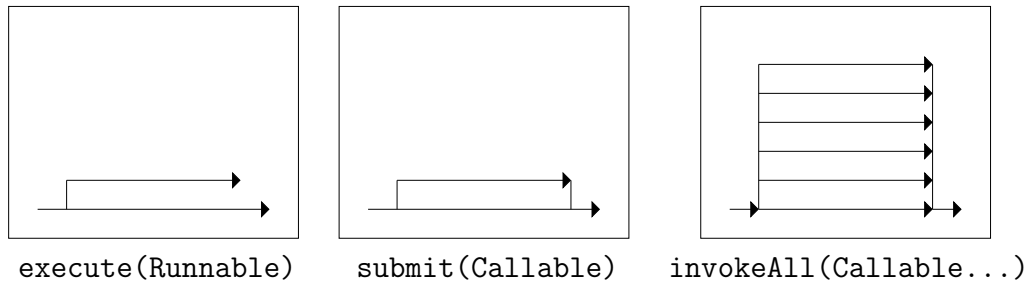


Figure 3: `execute` starts a result-less task without blocking. `submit` starts a task without blocking and returns a `Future` object which contains the result upon completion. `invokeAll` executes a number of tasks and blocks until a list with the results is returned.

5 Thread Pools

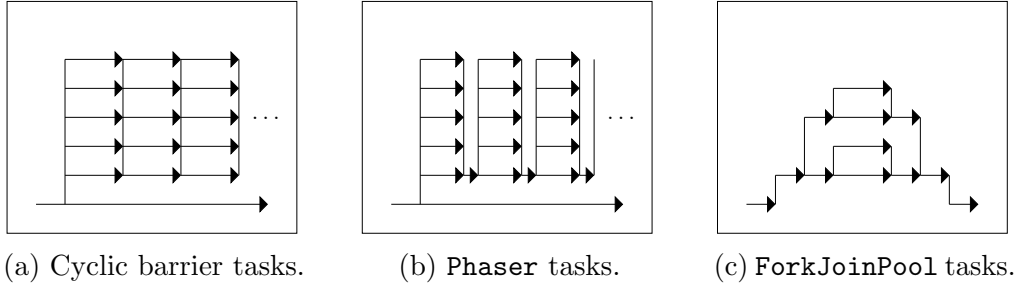
A *thread pool* is a transparent way to execute tasks asynchronously and is mostly used for data parallelism. It provides a number of *worker threads* and manages the distribution of *jobs* to them. The reuse of worker threads minimizes the overhead which would be caused by manually creating and starting threads.

With `ExecutorServices` Java defines an easy to use interface for executing parallel tasks. `Executors.newCachedThreadPool()` creates such a service with the number of threads equal to the number of available *CPUs*. The interface provides methods to invoke single or multiple tasks which are explained in Figure 3. The interface `Callable` defines a task that has a return value. This value is returned as `Future` object which holds the result when the task is complete. If an attempt is made to obtain the value earlier the calling thread waits until the completion.

For tasks that need to be executed periodically a *cyclic barrier* can be used. A cyclic barrier runs a set of tasks in parallel, awaits their completion, and then runs them again until a certain condition is met. When tasks should run with a defined pause between two cycles and without a barrier `Executors.newScheduledThreadPool(int)` with the number of worker threads can be used.

The class `Phaser` is a cyclic barrier that counts phases. Thread management has to be done manually and each thread must be registered on the `Phaser`. Phase numbers can be used to terminate execution after a given number of iterations by overwriting the method `onAdvance(int, int)`. Whenever a phase is complete `onAdvance(int, int)` is called with the current phase number and the number of registered threads on the `Phaser`. If the method returns `false` the `Phaser` is terminated. Appendix A shows how to implement a parallel particle swarm optimization using a `Phaser`.

☀ Java 7



6 ForkJoinPool

The class `ForkJoinPool` (FJP) allows easy implementation of *divide and conquer* algorithms in parallel. This class of algorithms splits input data into smaller sub-problems which are independent from each other and can be assumed to be data parallel. Unlike other thread pools the user cannot set the thread count directly in a FJP but can ask for a certain degree of parallelism, meaning that the pool maintains the given number of active worker threads. To fully utilize available CPUs only one FJP per application is therefore needed and the parallelism should not be set larger than the number of available cores to avoid unnecessary *context switch* overheads.

☀ Java 7

Work-stealing is used to prevent threads from waiting for work. Every worker thread has its own queue of tasks to execute. When the personal queue is empty, a worker picks a random task from another workers queue. This can be done *lock-free* with a native compare and set method. Furthermore the FJP periodically checks for stalled worker threads (sleeping, awaiting *I/O*, etc.) and starts new worker threads when needed [Lea11]. With those techniques every worker thread is either permanently active or gets compensated by a new thread.

A FJP executes `ForkJoinTasks` that can *fork* and later *join* (i.e. ‘wait’ for their completion) other `ForkJoinTasks`. In order to make divide and conquer algorithms parallel the subclass `RecursiveAction` can be used which provides an `invokeAll` method to substitute the recursion. This method executes at least one task in the current thread, building up the stack. When the problem size of a task is small enough it can be calculated directly [Bru11]. Appendix B gives an example of parallel matrix multiplication.

Benchmarks

A parallel matrix multiplication is used to analyze the performance of the FJP. As seen in Figure 4, differently sized matrices are multiplied with various degrees of parallelism. The single worker case is replaced by a normal matrix multiplication without the use of a FJP. While having a significant overhead with small matrix sizes FJPs work very well with large matrices gaining a *speedup* of approximately 10 with 8 workers. Enforcing a degree of parallelism greater than the number of available cores leads to worse performance due to context switches between threads. Since `ForkJoinTasks` have to be allocated on the heap using a FJP leads to a larger memory footprint than without using it.

7 Outlook

With the upcoming version 8 of Java lambda methods implementing interfaces with only one method are introduced. While this has no direct impact on concurrency it allows for less code lines to define tasks and needs no creation of actual instances of those classes by using the `invokedynamic byte-code` instruction [Goe12c]. Furthermore with the introduction of lambda methods the package `java.util.streams` is created to support *high-order functions* on lists [Goe12b]. With the help of FJPs and the new `Splititerator`, which splits a stream in smaller chunks, those streams can be processed in parallel [Goe12a].

Taking even a step further recent project like Rootbeer [PS12] and Aparapi [Fro12] transform java byte-code into CUDA code thus letting a user compute highly parallel on the *GPU* without switching the programming language. With Project Sumatra [Coo12] this functionality will eventually be implemented on the JVM.

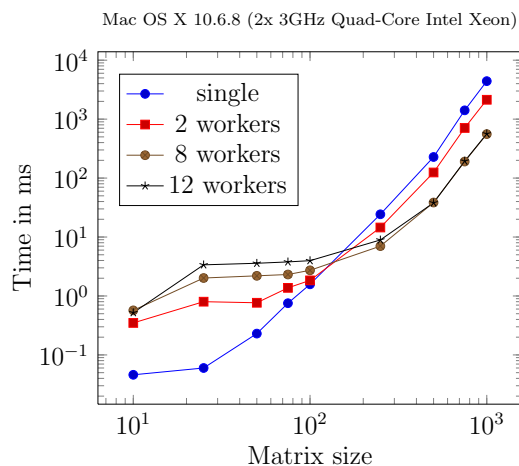


Figure 4: Benchmark results.

References

- [Bar12] Barker, M.: Arithmetic Overflow and Intrinsics. <http://bad-concurrency.blogspot.de/2012/08/>, August 2012.
- [Bru11] Bruno, E.: Using JDK 7's Fork/Join Framework. <http://www.drdobbs.com/jvm/using-jdk-7s-forkjoin-framework/231000556>, June 2011.
- [Coo12] Coomes, J.: Project Sumatra. <http://mail.openjdk.java.net/pipermail/announce/2012-September/000135.html>, September 2012.
- [Fro12] Frost, G.: Aparapi. <http://code.google.com/p/aparapi/>, July 2012.
- [Goe12a] Goetz, B.: ParallelPipeline.java. <http://hg.openjdk.java.net/lambda/lambda/jdk/file/5cd4d0c8b161/src/share/classes/java/util/streams/ParallelPipeline.java>, September 2012.
- [Goe12b] Goetz, B.: State of the Lambda: Libraries Edition. <http://cr.openjdk.java.net/~briangoetz/lambda/collections-overview.html>, April 2012.
- [Goe12c] Goetz, B.: Translation of Lambda Expressions. <http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html>, April 2012.

- [Lea08] Lea, D.: AtomicInteger.java. <http://www.docjar.com/html/api/java/util/concurrent/atomic/AtomicInteger.java.html>, April 2008.
- [Lea11] Lea, D.: ForkJoinPool.java. <http://grepcode.com/file/repo1.maven.org/maven2/org.codehaus.jsr166-mirror/jsr166y/1.7.0/jsr166y/ForkJoinPool.java>, July 2011.
- [Ora11a] Oracle: Java SE 7 Features and Enhancements. <http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html>, July 2011.
- [Ora11b] Oracle: Memory Consistency Properties. <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility>, July 2011.
- [PS12] Pratt-Szeliga, P.: Rootbeer. <https://github.com/pcpratts/rootbeer1/>, July 2012.
- [Sta09] StackUnderflow: Concurrency vs Parallelism – What is the difference? <http://stackoverflow.com/questions/1050222/>, June 2009.

A Parallel Particle Swarm Optimization

```

1 Phaser phaser = new Phaser() {
2     @Override
3     protected boolean onAdvance(final int phase, final int registeredParties) {
4         computeOptimum(particles);
5         return phase >= maxIter || registeredParties == 0;
6     }
7 };
8 phaser.register(); // ensures that the phaser is not terminated during initialization
9 for (final Particle p : particles) {
10     phaser.register(); // register the worker thread
11     new Thread() {
12         @Override
13         public void run() {
14             do {
15                 calculateNextPosition(p);
16                 phaser.arriveAndAwaitAdvance();
17             } while (!phaser.isTerminated());
18         }
19     }.start();
20 }
21 phaser.arriveAndDeregister(); // the current thread can deregister after initialization

```

Using a Phaser to implement a particle swarm optimization. For each particle, a thread is created to calculate its movement w.r.t. the current optimum. On advancing to the next phase the new optimum of the particles is computed.

B Parallel Matrix Multiplication

```
1 Matrix result = new Matrix(size);
2 new ForkJoinPool(parallelity).invoke(new MulTask(a, b, result, 0, 0, size, size));
3
4 class MulTask extends RecursiveAction {
5     public MulTask(final Matrix a, final Matrix b, final Matrix result,
6                     final int row, final int col, final int sizeX, final int sizeY) {
7         // assign fields
8     }
9     private void simple() {
10         for (int x = 0; x < sizeX; ++x) {
11             for (int y = 0; y < sizeY; ++y) {
12                 int r = 0;
13                 for (int z = 0; z < size; ++z) {
14                     r += a.get(row + x, z) * b.get(z, col + y);
15                 }
16                 result.set(row + x, col + y, r);
17             }
18         }
19     }
20     @Override
21     protected void compute() {
22         if (Math.min(sizeX, sizeY) <= THRESHOLD) {
23             simple();
24             return;
25         }
26         int asizeX = sizeX / 2, bsizeX = sizeX - asizeX;
27         int asizeY = sizeY / 2, bsizeY = sizeY - asizeY;
28         invokeAll(new MulTask(a, b, result, row, col, asizeX, asizeY),
29                 new MulTask(a, b, result, row + asizeX, col, bsizeX, asizeY),
30                 new MulTask(a, b, result, row, col + asizeY, asizeX, bsizeY),
31                 new MulTask(a, b, result, row + asizeX, col + asizeY, bsizeX, bsizeY));
32     }
33 }
```

Parallel divide and conquer matrix multiplication with a `ForkJoinPool`. The result matrix is split up in the `compute()` method until the size of a sub-matrix is below a certain `THRESHOLD`. Then the actual multiplication is performed in `simple()`.