Engineering Concurrent Library Components

Doug Lea
SUNY Oswego
Outline

Why does the code in java.util.concurrent look so different than that in an academic paper?

- Programming on layered, virtualized substrates
  - Performance anomalies are common
  - Libraries like j.u.c are “middle” members of substrate
  - Focus on Java, but many issues generalize

- Sample sets of problems and solutions
  - Adaptive waiting
  - Fine-grained memory access ordering
  - Memory placement and contention
  - Applications to bulk parallel operations

- Illustrate some general issues, some hacks
  - Performance impact can range from 1000X down to 1%
    - Usually don't know range in advance in any given case
Warmup: Sample Issues

Some questions without simple answers:

- How long does it take to block/unblock a thread?
- How long to re-read a variable inside a spin-wait?
- How to ensure you {always, never} re-read a field?
- How to minimize load $\rightarrow$ CAS window?
- How to isolate “thread local” variables?
- How to ensure that unused objects are reclaimable?
- How to ensure a method is compiled (not interpreted)
java.util.concurrent

- Atomic Variables
  - supporting compareAndSet (CAS), striped forms, etc
- Locks
  - including Conditions, ReadWriteLocks
- Synchronizers
  - Semaphores, barriers, etc
- Data Exchange
  - Queues, etc
- Concurrent Collections
  - Maps, Sets, Lists serving as shared resources
- Executor Frameworks
  - Tasks, Thread pools, Futures, work-stealing, completions...
Layered, Virtualized Systems

Lines of source code make many transitions on their way down layers, each imposing unrelated-looking ... 

- policies, heuristics, bookkeeping

... on that layer's representation of ...

- single instructions, sequences, flow graphs, threads

... and ...

- variables, objects, aggregates

One result: Poor mental models of the effects of any line of code
Hardware Trends

Opportunistically parallelize anything and everything

- More gates → More parallel computation
- Dedicated functional units, multicores
- More communication → More asynchrony
- Async (out-of-order) instructions, memory, & IO

One view of a common server
Interfaces

- Interface-like constructions hide implementations
  - Often get only one shot to define API at each layer

- Tension between Over- and Under- Abstraction
  - APIs impose consistency, composability constraints
  - More specialized APIs/components make fewer implementation tradeoffs
    - But require more specialized usages
    - Can lead to too many ways of doing almost the same thing

- Example: j.u.c.Queue extends java.util.Collection
  - Improves learnability; simplifies most usages
  - But requires that all implementations somehow allow removal of elements not at head of queue
  - Arbitrary removal is an unnatural act for many queue implementations, so can add overhead to “unrelated” methods

[See my PLDI 2012 talk for more on interactions with language features, etc]
Some Sources of Anomalies

- **Fast-path / slow-path**
  - “Common” cases fast, others slow
  - Ex: Caches, hash-based, JITs, exceptions, net protocols
  - Anomalies: How common? How slow?

- **Lowering representations**
  - Translation need not preserve expected performance model
    - May lose higher-level constraints; use non-uniform emulations
  - Ex: Task dependencies, object invariants, pre/post conds
  - Anomalies: Dumb machine code, unnecessary checks, traps

- **Code between the lines**
  - Insert support for lower-layer into code stream
  - Ex: VMM code rewrite, GC safepoints, profiling, loading
  - Anomalies: Unanticipated interactions with user code
Leaks Across Layers

Higher layers may be able to influence policies and behaviors of lower layers

- **Sometimes control is designed into layers**
  - Components provide ways to alter policy or bypass mechanics
    - Sometimes with explicit APIs
    - Sometimes the “APIs” are coding idioms/patterns
    - Ideally, a matter of performance, not correctness
  - Underlying design issues are well-known
    - See e.g., Kiczales “open implementations” (1990s)
  - Leads to eat-your-own-dog-food development style

- **More often, control arises by accident**
  - Designers (defensibly) resist specifying or revealing too much
    - Sometimes even when “required” to do so (esp hypervisors)
  - Effective control becomes a black art
    - Fragile; unguaranteed byproducts of development history
Theme: Data-Parallel Composition

Tiny map-reduce example: sum of squares on array

- Familiar sequential code/compilation/execution
  
  ```
  s = 0; for (i=0; i<n; ++i) s += sqr(a[i]); return s;
  ```

  ... or ...
  
  ```
  reduce(map(a, sqr), plus, 0);
  ```

- May be superscalar even without explicit parallelism

- Parallel needs algorithm/policy selection, including:
  
  - Granularity: #cores vs task overhead vs memory/locality
  - Reduction: Tree joins? Async completions?

- Results in families of code skeletons

  - Some of them are even faster than sequential
Bulk Operations and Amdahl's Law

- Sequential set-up/tear-down limits speedup
  - Or as lost parallelism = (cost of seq steps) * #cores
  - Can easily outweigh benefits

- Can parallelize some of these
  - Recursive forks
  - Async Completions
  - Adaptive granularity
    - Best techniques take non-obvious forms
    - Some rely on nature of map & reduce functions

- Cheapen or eliminate others
  - Static optimization
    - Jamming/fusing across operations; locality enhancements
  - Share (concurrent) collections to avoid copy / merge
class SumSqTask extends RecursiveAction {
    final long[] a; final int l, h; long sum;
    SumSqTask(long[] array, int lo, int hi) {
        a = array; l = lo; h = hi;
    }
    // (One basic form; many improvements possible)
    protected void compute() {
        if (h - l < THRESHOLD) {
            for (int i = l; i < h; ++i)
                sum += a[i] * a[i];
        }
        else {
            int m = (l + h) >>> 1;
            SumSqTask rt = new SumSqTask(a, m, h);
            rt.fork(); // pushes task
            SumSqTask lt = new SumSqTask(a, l, m);
            lt.compute();
            rt.join(); // pops/runs or helps or waits
            sum = lt.sum + rt.sum;
        }
    }
}

Tediously similar code for many other bulk operations
Case 1: Adaptive Waiting

Goal: When waiting for a change in a variable that is outside of local ability to modify, choose scheme to:

- Maximize **throughput**, and/or minimize **latencies** or their **variance** (fairness), and/or minimize **interference** (impact on other activities), and/or minimize **energy** requirements

- **Impossible in general**
  - Among main motivations for non-blocking approaches
    - But waiting often unavoidable
  - **Huge performance impact for some bad decisions**
    - The same issues arise at multiple layers → 1000X variance

- **Classic push/pull tradeoffs**
  - Spin polling for change, vs “do nothing” until signalled
  - Best-practice resolution: transition from pull to push
    - j.u.c policy: Always eventually do so (no unbounded spins)
Spinning

Obvious form:

```java
volatile boolean cond; …
while (!cond) { /* skip */ }
```

Some problems:

- Processor may detect as an idle-loop and power down core
- Processor may combine/elide instructions that overwhelm memory controller, so act superscalar. Or the opposite.
- Memory system may be saturated, slowing down other threads, especially if cache line shared with other variables
- JVM may insert execution counter and/or safepoint check into loop → more coherence traffic
- OS and/or VMM may context-switch out thread, thus blocking when user code expects spinning
- No transition mechanics to blocking forms
Blocking

Obviously wrong form:

\[ \text{T1: if (!cond) suspend();} \]
\[ \text{T2: cond = true; t1.resume();} \]

- Each layer must deal with suspend-resume race
  - Usually, internal semaphore-like mechanics
    - Java intrinsics: `LockSupport.{park, unpark}`
    - Park returns immediately if semaphore set, so a near no-op
      - Sometimes set “accidentally”, so calls must use `while`, not `if`

- Each layer feels free to pile on bookkeeping
  - Reasoning: “the thread isn't doing anything useful anyway”
  - Even for signalling/resumption (ignoring Amdahl's law)

- Sometimes leads to fall-off-cliff discontinuities
  - Can be hundreds of thousands of cycles to block/unblock
Transition Policies

Goal: Spin a while, then block

- Common ideas/approaches:
  - Backoffs: Do “nothing” for increasing periods before block
    - If “nothing” != spinning, entails blocking + OS timer wakeups
      - Timer granularity too coarse for many uses
  - Bound the number of cycles possibly wasted spinning, to guarantee energy/interference bounds
    - Decision is arbitrary, but users want a “good” choice
  - Rate Predictive: If average wait time < block+unblock time, then only block if (sufficiently) past average
    - Entails statistical decision mechanics & randomization

- Information required for implementing these (and others) spans layers, and/or is unreliable
- Requires check-then-act decisions based on stale data
Spin Implementations

- No single implementation in j.u.c
  - Some mechanics shared in AbstractQueuedSynchronizer
  - Many instead use specializations of common patterns
    - Sometimes inconsistently; only improved when updated

- Some common ingredients
  - Avoid loops of form \( b: \text{load cond}; \text{branchIfZero } b; \)
    - Usually achieve more uniform rate with less uniform branches
    - If nothing better, generate simple random numbers inside loop – often Marsaglia XorShift (cheap and good enough)
      - Test some bits to randomize control paths
  - Add one or two voluntary context switches (Thread.yield)
    - A rough approximation to early blocking if others runnable
    - May avoid some of the heavier blocking mechanics
    - Thread.yield spec is very weak, so this is very heuristic
Bounding Spins

- **Time-based**
  - Calling clock (Java nanoTime) generates memory traffic
    - May be heavy traffic because of clock monotonicity specs
    - May also cause safepoint and context switch
  - Using on-chip cycle counters doesn't work if thread moved etc

- **Count-based**
  - Interpreted vs compiled vs optimized loops run at different rates
  - Statically bounded may have more predictable performance
    - May avoid JITs inserting instrumentation and safepoints
  - Randomized bounds may improve fairness
    - May avoid getting OS scheduler stuck in patterns

- **Combinations**
  - No single best tradeoff across all the may's
  - Count-based, with time-based secondary checks, works
    - But normally delay even the first call to get time
Spin → Block Thresholds

**Estimation**

- **Static**: Find threshold associated with max target wastage on common machines for highly contended use cases
  - Pick compromise value good enough across platforms
- **Dynamic**: Adjust static estimate to improve throughput by tracking loop counts/times
  - But adjustments tend to stabilize at max value
    - Not often worthwhile compared to instrumentation overhead

**Semantics-dependent add-ons, including**

- **Queues**: Block sooner if head of queue already blocked
- **Allow “barging” for lock-like sync**
  - Let another polling thread get lock instead of the one signalled
  - Signallee then restarts entire process with lower threshold
  - Improves throughput: minimize time lock not held but wanted
  - But weakens fairness (latency variance)
Interspersion

Integrate wait mechanics into primary functionality

- Recast most functionality as state-machine loop
  - Reduces total number of safepoints etc
- Enables some attempts at helping rather than waiting
  - Data structure maintenance
- Can meet other API requirements while spinning
  - Checking Interrupts and InterruptedExceptions
  - Support for monitoring and management
- If barrier-like: Help with wakeups if no wait required
  - Helps parallelize signalling

Most obvious result: Messy code
Case 2: Memory Access Ordering

- **Java (also C++, C) Memory Model for locks**
  - Sequentially Consistent (SC) for data-race-free programs
    - A requirement for implementations of locks and synchronizers
- **Java volatiles (and default C++ atomics) also SC**
  - Generate fence instructions and/or compiler constraints
    - (Some) can be CAS’ed, and used to implement locks
- **Interactions with non-volatile accesses complicated**

First approximation of reordering rules:

<table>
<thead>
<tr>
<th>1st/2nd</th>
<th>Plain load</th>
<th>Plain store</th>
<th>Volatile load</th>
<th>Volatile store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain load</td>
<td></td>
<td></td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Plain store</td>
<td></td>
<td></td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Volatile load</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Volatile store</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

**Plus interactions with dependencies, causality, etc**
Relaxed Atomics

- Provide intermediate ordering control
  - May greatly reduce overhead in performance-critical code
    - But must consider r/w before and after library method call
    - One approach is to view in terms of invalidations (next slide)
  - Even more complex interactions with plain & volatile access

- C++ standardized access modes (acquire, release, ...)

- Java: Unstandardized JVM “internal” intrinsics
  - Ideally, a bytecode for each mode of (load, store, CAS)
    - Would fit with No L-values (addresses) Java rules
  - Instead, intrinsics take object + field offset arguments
    - Establish on class initialization, then use in Unsafe API calls
    - Non-public; truly “unsafe” since offset args can't be checked
      - Can be used outside of JDK using odd hacks if no security mgr
      - j.u.c supplies public wrappers that interpose (slow) checks
Publication and Transfers

Weaker protocols avoid more invalidation

Class X { int field; X(int f) { field = f; } }

- For shared var v (other vars are locals):
  P: p.field = e; v = p; ||
  C: c = v; f = c.field;

- Use weakest construction still ensuring that C:f is usable, considering (among other issues):
  - “Usable” can be algorithm- and API-dependent
    - Consistency with reads>writes of other shared vars?
  - Is write to v final (i.e., the last-ever write)? including:
    - Write Once (null → x), Consume Once (x → null)
    - Is write to x.field final?
    - Is there a unique uninitialized value for field?
Example: Transferring Tasks

- Work-stealing Queues perform ownership transfer
  - Push: make task available for stealing or popping
    - Needs lightweight store-fence
  - Pop, steal: make task unavailable to others, then run
    - Needs CAS with at least acquire-mode fence

```
T1: push(w) --
    w.state = 17;
    slot = q;

T2: steal() --
    w = slot;
    if (CAS(slot, w, null))
        s = w.state; ...
```

Task w
Int state;

Queue slot

Requires: s == 17
Example: ForkJoinTask.fork API spec

“Arranges to asynchronously execute this task. While it is not necessarily enforced, it is a usage error to fork a task more than once unless it has completed and been reinitialized. Subsequent modifications to the state of this task or any data it operates on are not necessarily consistently observable by any thread other than the one executing it unless preceded by a call to join() or related methods, or a call to isDone() returning true.”

- The no-refork rule ultimately reflects internal relaxed consistency mechanics based on ownership transfer
  - The mechanics leverage fact that refork before completion doesn't make sense anyway
- The inconsistent-until-join rule reflects arbitrary state of, e.g., the elements of an array while it is being sorted
  - Also enables weaker ordering (more parallelism) while running
- Would be nicer to statically enforce
  - Secretly, the no-refork rule cannot now be dynamically enforced
Composition and Consistency

- Consistency policies are intrinsic to systems with multiple readers or multicast (so: part of API design)
- Most consistency properties do not compose
- IRIW Example: \( \text{vars } x, y \text{ initially } 0 \rightarrow \text{events } x, y \text{ unseen} \)
  - Activity A: send \( x = 1 \); \hspace{1cm} // (multicast send)
  - Activity B: send \( y = 1 \);
  - Activity C: receive \( x \); receive \( y \); \hspace{1cm} // see \( x=1, y=0 \)
  - Activity D: receive \( y \); receive \( x \); \hspace{1cm} // see \( y=1, x=0 \) ? Not if SC
- For vars, can guarantee sequential consistency
  - JMM: declare \( x, y \) as volatile
- Doesn't necessarily extend to component operations
  - e.g., if \( x, y \) are two maps, & the \( r/w \) operations are \( \text{put}/\text{get}(k) \)
- Doesn't extend at all under failures
  - Even for fault-tolerant systems (CAP theorem)
Optimization and Ordering

- Orderings inhibit common compiler optimizations
  - Inhibiting wrong ones may also inhibit those you want

- Requires “manual” dataflow optimizations
  - Manually hoisting reads, exception & indexing checks, etc
  - Manually inlining to avoid call opaqueness effects
    - JIT inlining rules usually not too smart here
  - Can be challenging to express in source code
    - Some resort to other intrinsics to bypass redundant checks

- Use odd loop forms to better position safepoints etc
  - Example: increment x.field by: int c;
    do {} while (!CAS(x, fieldOffset, c = x.field, c+1));
  - Usually better than other forms because field read more likely to be adjacent to CAS, so smaller race window
Coping with *Idiot Savant* Compilers

A sampling of tiny, sometimes transient, issues:

- **Dispatching/Inlining**
  - Recasting code to save a few bytecodes in front-end compiled form improves chances of inlining
  - So, e.g., using Assert statements may impede inlining
  - Inlining only fast paths to funnel slower dispatch to single virtual call points

- **Avoiding or exploiting Invisible code**
  - Rely on default-zero/null initialization
  - Leverage cases where null-checks are required anyway
    - Null is almost always the best “special” value to check
  - Minimize boxing, class-loading, synthetic access methods, etc

- **Compilation Plans**
  - Forcing code warmups combats obliviousness to Amdahl's law for sequential bottlenecks
  - Avoiding decompilation, “rare” traps, JNI, and other slow stuff
Determinism à la carte

- Common components entail algorithmic randomness
  - Hashing, skip lists, crypto, numerics, etc
    - Fun fact: The Mark I (1949) had hw random number generator
  - Visible effects; e.g., on collection traversal order
    - API specs do not promise deterministic traversal order
      - Bugs when users don't accommodate
  - Randomness more widespread in concurrent components
    - Adaptive contention reduction, work-stealing, etc

- Plus non-determinism from multiple threads
  - Visible effects interact with consistency policies

- Main problem across all cases is bug reproducibility
  - A design tradeoff across languages, libraries, and tools
  - Non-deterministic performance bugs exist independently
Performance Testing

Need continuous testing during development

- Mainly **Abstract Use Cases** (also actual use cases)
  - Producer/consumer, shared maps, ... little/no user computation
- Randomize or vary control over confounding factors
  - Main challenge is to identify these factors
    - Warm-up, memory placement, compilation plans, ...
    - JVM switches: biased locking, memory, GC algorithms, ...
    - Run on multiple platforms (test machines)
- Avoid dead code and micro-benchmark artifacts
  - Incorporate correctness checks (simple cheap invariants)
  - Randomize computation to avoid optimizing checks away
- Avoid scheduling artifacts – e.g., occasionally force full barriers
- Use appropriate metrics (throughput, latency, variance, etc)
  - Often choose more predictable performance over fastest
  - Simple visual monitoring (perfbar etc) often most helpful
Case 3: Memory Placement

Memory contention and locality effects
- False-sharing, NUMA, etc can have huge impact
  - Reduce parallel progress to memory system rates

Some GC mechanics worsen impact; esp card marks
- When writing a reference, JVM also writes a bit/byte in a table indicating that one or more objects in its address range (often 512 bytes wide) may need GC scanning
- The card table becomes highly subject to contention
- Yang et al (ISMM 2012) report 378X slowdown

JVMs cannot allow precise object placement control
- Must reserve right to move objects dynamically
  - Production JVMs normally initially allocate into thread-local arenas, and may then move to shared spaces
  - JVM/GC oblivious to higher-level locality constraints
    - Including “ThreadLocal”!
Spreading Out Variables

- No standardized Java version of C padding idioms
  - (C padding idioms are rarely guaranteed either)
  - JDK8 @Contended annotation will help

- Non-portable hacks
  - Adding unused fields in objects
  - Over-allocating arrays and skipping elements

- Worthwhile in components expected to hit heavy memory contention but highly heuristic/fallible
  - Don't know how many bits in a reference/pointer
  - Don't know platform coherence unit (usu cache-line) width
  - Don't know object layout/packing rules
    - Most pack in decreasing size, secondarily in declaration order
  - Too much spread can interfere with locality/affinity
Reclamation

- Allocation often used inside non-blocking code
  - Using pointers avoids ABA CAS issues by relying on GC
  - But production GCs are themselves **NOT** fully non-blocking
    - So no j.u.c code with allocation is provably wait- or lock- free
- Can cause bloat; esp “floating garbage”
  - Concurrent GC can miss noticing that an object is no longer referenced, causing its transitive closure to stay alive
  - If these links form a linear chain, also slows down GC
- Regularly detaching references avoids worst effects
  - Example: Michael & Scott – style queues (next slide)
    - **Self-linking** next fields when unused
  - Techniques similar to those in non-GC contexts
ConcurrentLinkedQueue

- Extend Michael & Scott Queue (PODC 1996)
  - CASes on different vars (head, tail) for put vs poll
  - If CAS of tail from t to x on put fails, others try to help
    - By checking consistency during put or take
  - Restart at head on seeing self-link

1: CAS head from h to n
2: self-link h

Put x

1: CAS t.next from null to x
2: CAS tail from t to x

Poll
Summary

- Performance anomalies are common on layered, virtualized platforms
  - The problems are general
    - Fast-path/slow-path, invisible code, lowering, API constraints, ...
  - The details are quirky
    - Performance impact of black art not likely to subside
    - When lower layers “fix” one anomaly, another may appear
- Greatest impact on heavily used library components
  - Sometimes hard to find the algorithm in the code
  - These are worth the effort to micro-optimize
    - And yet still release more-or-less readable source code
  - Hard to predict properties of fully implemented component using a given algorithm
    - An obstacle for adapting new algorithms
Backup slides

Backup slides follow
Avoiding Invalidation on Writes

Avoid: storeFence; \( v = x; \) storeLoadFence

- **Static single or final Write**
  - Single thread issuing final write is structurally determined
  - Example: storeFence; \( v = x; \)

- **Dynamic single or final Write**
  - Ensuring one writer requires distinguished value
  - Example: [storeFence] CAS(&\( v \), null, x)

- **Validated (including “double-checked”)**
  - Don't fence write if reads validate with CAS
  - Example: if (\( v == \) null) { … if (CAS(&\( v \), null, x) … }

- **Dependent**
  - Don't fence var if accesses nested under another
  - Example: lock; \( v = x; \) unlock;
Composition Using Injection

- Simplify data-parallelism by allowing injection of code snippets into holes in skeletons
  - Subject to further transformation/optimizations
  - Some users need to program the skeletons
    - Some only need to occasionally fine-tune them
  - Most users usually just want to supply the snippets
- Need to represent and manipulate code snippets
  - Closure-objects, lambdas, macros, templates, etc
    - Each choice has good and bad points
      - e.g., megamorphic dispatch vs code bloat
      - Easy to confuse the means and ends (lambda != FP)
  - Or push up one level and use generative IDE-based tools or layered languages
    - A long heritage for GUI, web page, etc composition of snippets
Usability of Abstractions

Users like and use some API styles more than others

**Futures**: \( r = \text{ex}.\text{submit}(\text{func}); \ldots; \text{use}(r.\text{get}()); \)

- Idea: parallel variant of lazy evaluation
  - Nicely extend to recursive parallelism (j.u.c ForkJoinTasks)
  - Intuitive/pleasant even if need explicit syntax to get result
    - But can be a resource management problem when recursively blocked on indivisible leaf actions (like IO)
      - Chains of blocked threads; requires internal mgt heuristics

**Completions**: \( t2 = \text{new CC}(1, t1); \ldots t2.\text{fork}(); \ldots \)

- Idea: arrange to trigger an action when other(s) complete
  - Atomic triggers for continuations avoid cascaded blocking
  - Often less intuitive/pleasant
  - Full support with ugly CountedCompleter API
  - Limited support with nicer CompletableFuture API
Top-down: Create a transactional (sub)language to support multi-operation, multi-object atomicity

- Many hard issues: contention, overhead, space bloat, side-effect rollback, etc
- So far, at best, highly variable performance

Library-based: Provide Collections supporting finite sets of possibly-compound atomic operations

- Example: ConcurrentHashMap.putIfAbsent
  - Key-value maps often the focus of transactions; cf SQL
  - Can be implemented efficiently

Improve atomic APIs based on experience

- e.g., adding computeIfAbsent, recompute
- Usually can only do so for implementations, not interfaces

But still no multi-object atomicity guarantees
Implementing Shared Data Structures

Mostly-Write
- Most producer-consumer exchanges
- Apply combinations of a small set of ideas
  - Use non-blocking sync via compareAndSet (CAS)
    - Or hardware TM if available
  - Relax internal consistency requirements
  - Reduce point-wise contention
  - Arrange that threads help each other make progress

Mostly-Read
- Most Maps & Sets
  - Structure to maximize concurrent readability
  - Without locking, readers see legal (ideally, linearizable) values
  - Often, using immutable copy-on-write internals
  - Apply write-contention techniques from there
Libraries and Software Process

- Incrementalism requires shorter cycles
  - Releasing a component easier than language or hw
  - Users: Trying new library easier than new language
  - Continuous feedback on functionality, usability

- New APIs provide ideas for restructuring programs
  - Balancing with what users say they want
    - Some ideas don't make it into release
  - Also provide new user bug opportunities
    - Has led to new bug pattern detectors in findBugs

- Multiple audiences
  - Heaviest j.u.c. users use it to build layered frameworks
  - Users with better ideas can create better components
Task-Based Parallel Evaluation

- Programs can be broken into tasks
  - Under some appropriate level of granularity
- Workers/Cores continually run tasks
  - Sub-computations are forked as subtask objects
- Sometimes need to wait for subtasks
  - Joining (or Futures) controls dependencies

```c
f() = {
    split;
    fork;
    join;
    reduce;
}
```
For recursive decomposition, deques arrange tasks with the most work to be stolen first. (See Blelloch et al for alternatives)

Example: method s operating on array elems 0 ... n:
Parallel Recursive Decomposition

Typical algorithm

```java
Result solve(Param problem) {
    if (problem.size <= THRESHOLD)
        return directlySolve(problem);
    else {
        in-parallel {
            Result l = solve(leftHalf(problem));
            Result r = solve(rightHalf(problem));
        }
        return combine(l, r);
    }
}
```

❖ To use FJ, must convert method to task object
❖ “in-parallel” can translate to invokeAll(leftTask, rightTask)
❖ The algorithm itself drives the scheduling
❖ Many variants and extensions