

---

# Parallelism From The Middle Out

(Adapted from PLDI 2012)

Doug Lea  
SUNY Oswego

# The Middle Path to Parallel Programming

- ◆ **Bottom up: Make computers faster via parallelism**
  - ◆ Instruction-level, multicore, GPU, hw-transactions, etc
    - ◆ Initially rely on non-portable techniques to program
- ◆ **Top down: Establish a model of parallel execution**
  - ◆ Create syntax, compilation techniques, etc
  - ◆ Many models are available!
- ◆ **Middle out: Encapsulate most of the work needed to solve particular parallel programming problems**
  - ◆ Create reusable APIs (classes, modules, frameworks)
    - ◆ Have both hardware-based and language-based dependencies
  - ◆ Abstraction á la carte

# The customer is always right?

- ◆ Vastly more usage of parallel library components than of languages primarily targeting parallelism
  - ◆ Java, MPI, pthreads, Scala, C#, Hadoop, etc libraries
- ◆ Probably not solely due to inertia
  - ◆ Using languages *seems* simpler than using libraries
  - ◆ But sometimes is not, for some audiences
- ◆ In part because library/language/IDE/tool borderlines are increasingly fuzzy
  - ◆ Distinctions of categories of support across ...
    - ◆ Parallel (high throughput)
    - ◆ Concurrent (low latency)
    - ◆ Distributed (fault tolerance)
  - ... also becoming fuzzier, with many in-betweens.

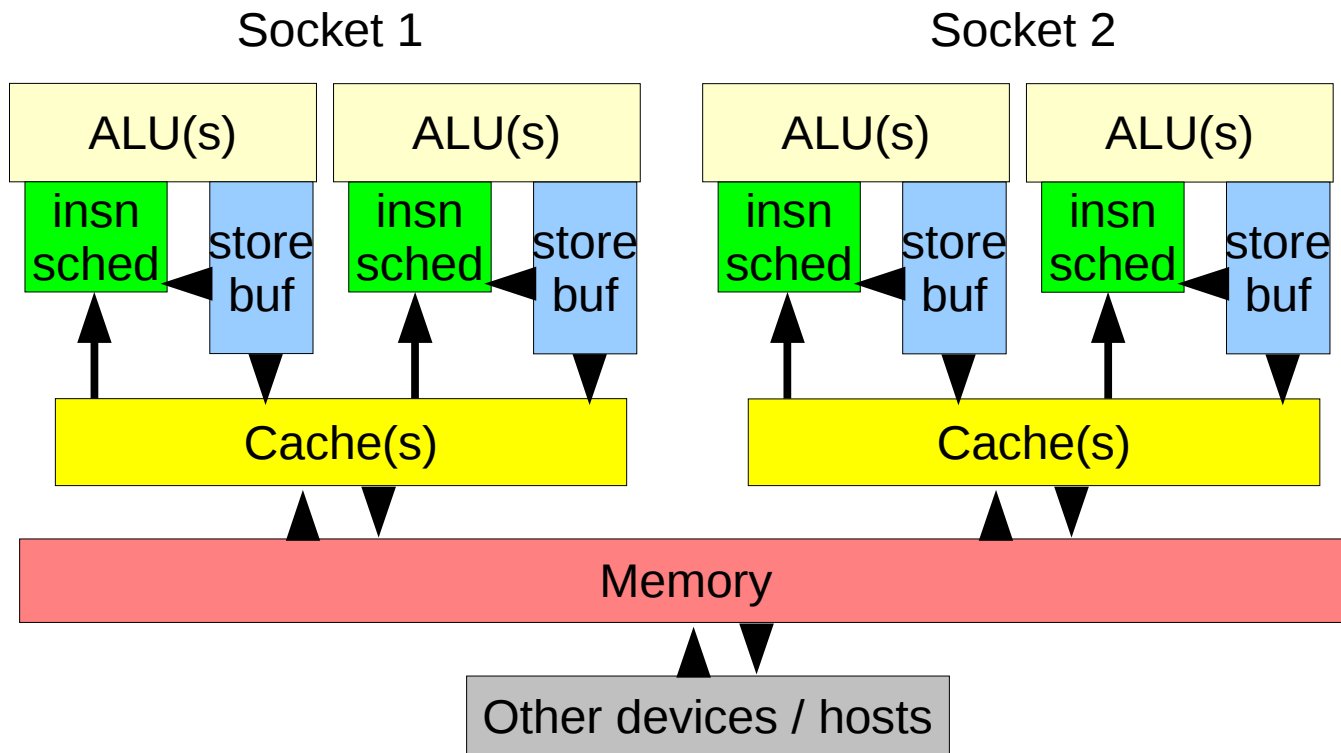
# Abstractions vs Policies

- ◆ **Hardware parallelism is highly opportunistic**
  - ◆ Directly programming not usually productive
- ◆ **Effective parallel programming is too diverse to be constrained by language-based policies**
  - ◆ e.g., CSP, transactionality, side-effect-freedom, isolation, sequential consistency, determinism, ...
    - ◆ But they may be helpful constraints in some programs
- ◆ **Engineering tradeoffs lead to medium-grained abstractions**
  - ◆ Still rising from the Stone Age of parallel programming
- ◆ **Need diverse language support for expressing and composing them**
  - ◆ Old news (Fortress, Scala, etc) but still many open issues

# Hardware Trends

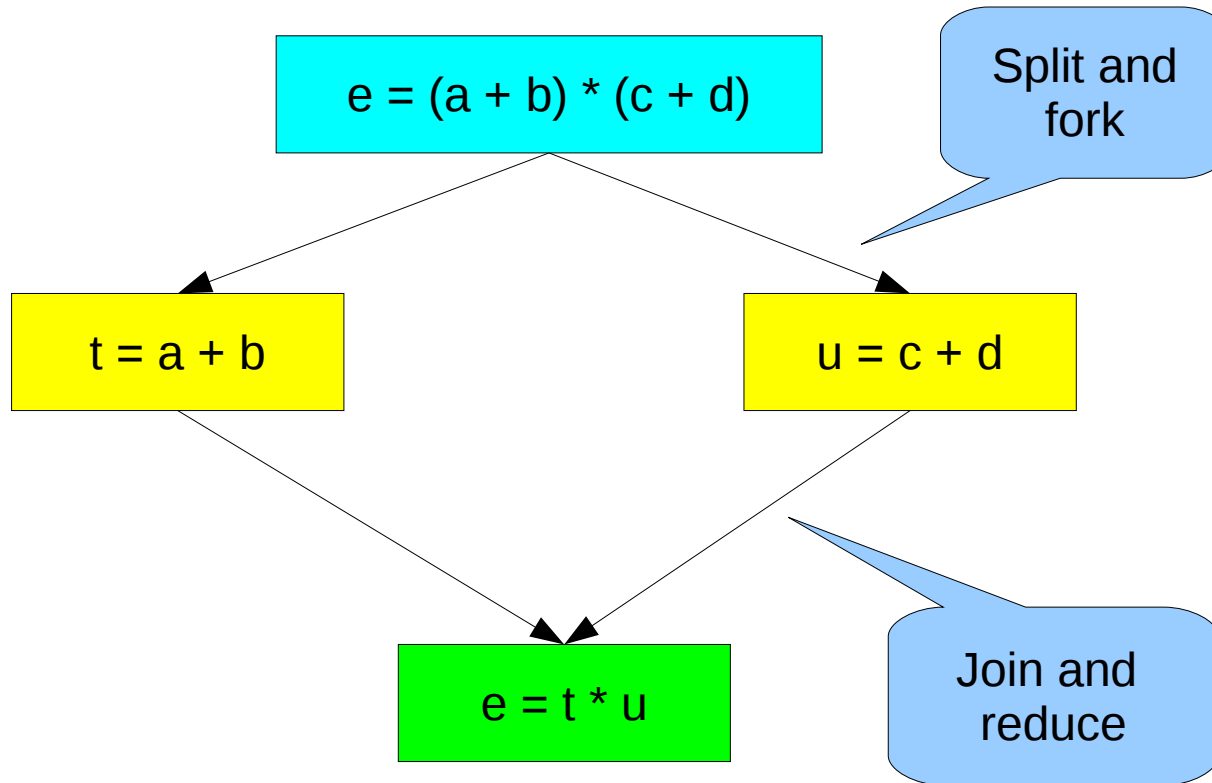
Opportunistically parallelize anything and everything

- ◆ 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

# Parallel Evaluation



Parallel divide and conquer

# Parallel Evaluation inside CPUs

- ◆ Overcome problem that instructions are in sequential stream, not parallel dag
- ◆ Dependency-based execution
  - ◆ Fetch instructions as far ahead as possible
  - ◆ Complete instructions when inputs are ready (from memory reads or ops) and outputs are available
    - ◆ Use a hardware-based simplification of dataflow analysis
- ◆ Doesn't always apply to multithreaded code
  - ◆ Dependency analysis is shallow, local
  - ◆ What if another processor modifies a variable accessed in an instruction?
  - ◆ What if a write to a variable serves to release a lock?

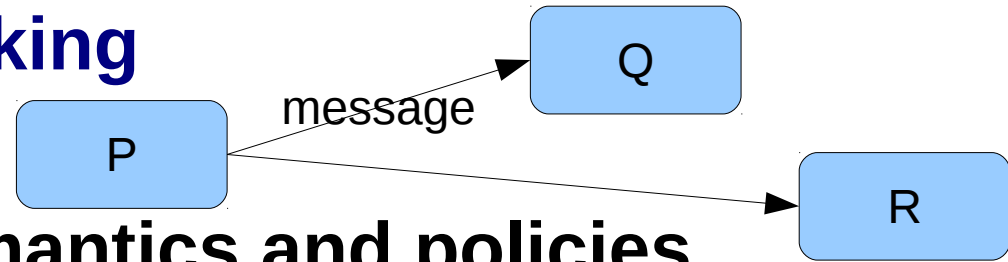
# Parallelism in Components

- ◆ Over forty years of parallelism and asynchrony inside commodity platform software components
  - ◆ Operating Systems, Middleware, VMs, Runtimes
    - ◆ Overlapped IO, device control, interrupts, schedulers
    - ◆ Event/GUI handlers, network/distributed messaging
    - ◆ Concurrent garbage collection and VM services
  - ◆ Numerics, Graphics, Media
    - ◆ Custom hw-supported libraries for HPC etc
- ◆ Result in better throughput and/or latency
  - ◆ But point-wise, quirky; no grand plan
  - ◆ Complex performance models. Sometimes *very* complex
- ◆ Can no longer hide techniques behind opaque walls
  - ◆ Everyday programs now use the same ideas



# Processes, Actors, Messages, Events

## Deceptively simple-looking



### ◆ Many choices for semantics and policies

- ◆ Allow both actors and passive objects?
  - ◆ Single- vs multi- threaded vs transactional actors?
  - ◆ One actor (aka, the event loop) vs many?
- ◆ Isolated vs shared memory? In-between scopes?
  - ◆ Explicitly remote vs local actors?
- ◆ Distinguish channels from mailboxes?
  - ◆ Message formats? Content restrictions? Marshalling rules?
- ◆ Synchronous vs asynchronous messaging?
- ◆ Point-to-point messaging vs multicast events?
  - ◆ Rate limiting? Consensus policies for multicast events?
- ◆ Exception, Timeout, and Fault protocols and recovery?

# Process Abstractions

- ◆ **Top-down:** create model+language (ex: **CSP+Occam**) supporting a small set of semantics and policies
  - ◆ Good for program analysis, uniformity of use, nice syntax
  - ◆ Not so good for solving some engineering problems
- ◆ **Middle-Out:** supply policy-neutral components
  - ◆ Start with the Universal Turing Machine vs TM ploy
    - ◆ **Tasks** – executable objects
    - ◆ **Executors** – run (multiplex/schedule) tasks on cores etc
    - ◆ Specializations/implementations may have little in common
  - ◆ Add synchronizers to support messaging & coordination
    - ◆ Many forms of **atomics, queues, locks, barriers**, etc
  - ◆ Layered frameworks, DSLs, tools can support sweet-spots
    - ◆ e.g., Web service frameworks, Scala/akka actors
    - ◆ Other choices can remain available (or not) from higher layers

# Libraries Focus on Tradeoffs

Library APIs are platform features with:

## ◆ Restricted functionality

- ◆ Must be expressible in base language (or via cheats)
- ◆ Tension between efficiency and portability

## ◆ Restricted scopes of use

- ◆ Tension between **Over- vs Under- abstraction**
  - ◆ Usually leads to support for many styles of use
  - ◆ Rarely leads to sets of completely orthogonal constructs
    - ◆ Over time, tends to identify useful (big & small) abstractions

## ◆ Restricted forms of use

- ◆ Must be **composable** using other language mechanisms
- ◆ Restricted usage syntax (less so in Fortress, Scala, ...)
  - ◆ Tensions: economy of expression, readability, functionality

# Layered, Virtualized Systems

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

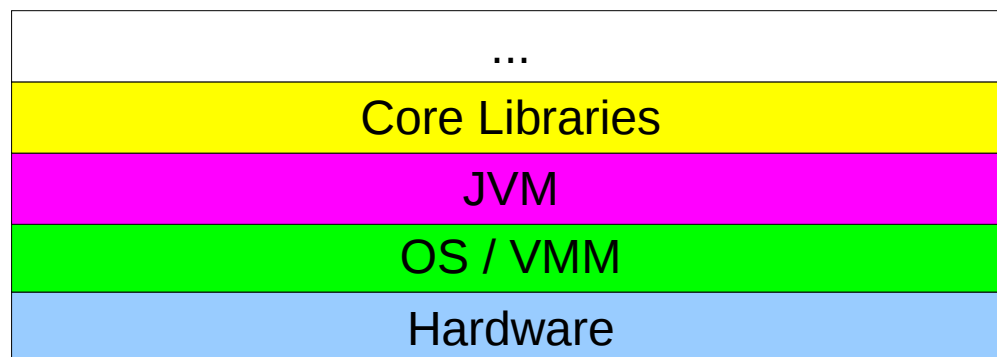
◆ **policies, heuristics, bookkeeping**

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

◆ **single instructions, sequences, flow graphs, threads**

... and ...

◆ **variables, objects, aggregates**



Each may entail internal layering

**One result:**

**Poor mental models of the effects of any line of code**

# 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

# Composition

## Components require language composition support

- ◆ APIs often reflect how they are meant to be composed
- ◆ To a first approximation, just mix existing ideas:
  - ◆ **Resource-based composition using OO or ADT mechanics**
    - ◆ e.g., create and use a shared registry, execution framework, ...
  - ◆ **Process composition using Actor, CSP, etc mechanics**
    - ◆ e.g., messages/events among producers and consumers
  - ◆ **Data-parallel composition using FP mechanics**
    - ◆ e.g., bulk operations on aggregates: map, reduce, filter, ...
- ◆ The first approximation doesn't survive long
  - ◆ Supporting multiple algorithms, semantics, and policies forces interactions
    - ◆ Requires integrated support across approaches

# 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:

◆ Split work: Static? Dynamic? Affine? Race-checked?

◆ Granularity: #cores vs task overhead vs memory/locality

◆ Reduction: Tree joins? Async completions?

◆ Substrate: Multicore? GPU? FPGA? Cluster?

## ◆ 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

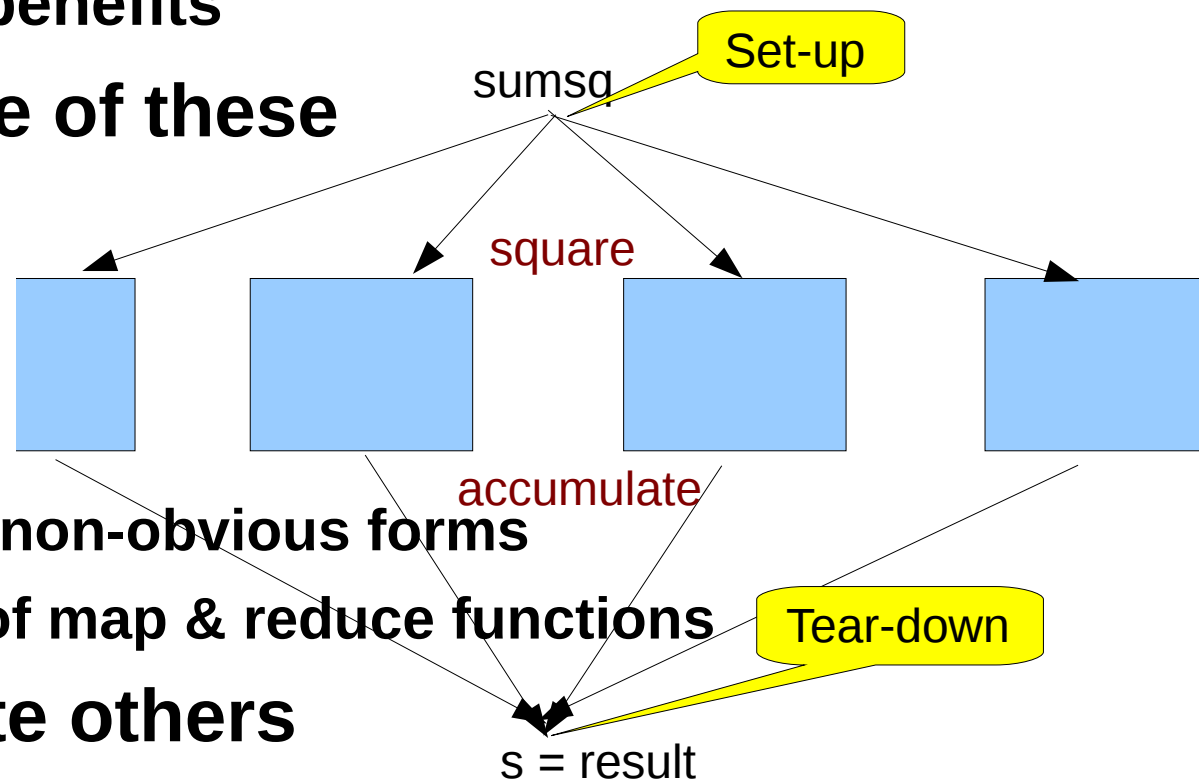
## Can parallelize some of these

- Recursive forks
- Async Completions
- Adaptive granularity

- Best techniques take non-obvious forms
- Some rely on nature of map & reduce functions

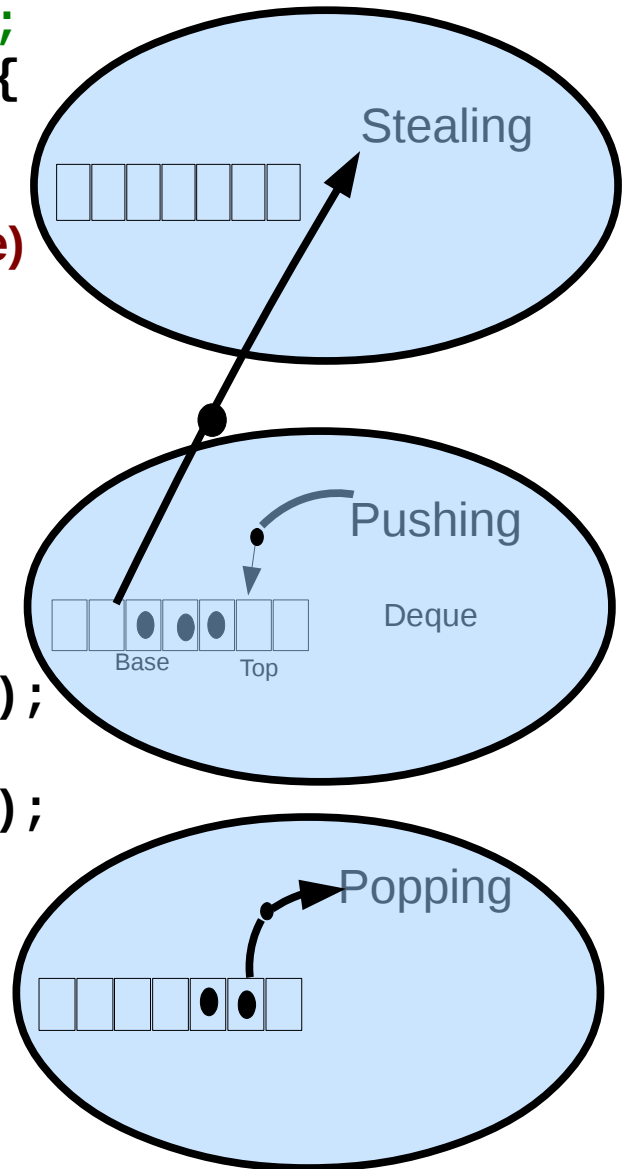
## Cheapen or eliminate others

- Static optimization
  - Jamming/fusing across operations; locality enhancements
- Share (concurrent) collections to avoid copy / merge



# Sample ForkJoin Sum Task

```
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

# 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 or DSL-like extensions or DSLs
    - ◆ A long heritage for GUI, web page, etc composition of snippets

# Composition on Shared Resources

- ◆ **Top-down:** Create a transactional (sub)language to support multi-operation, multi-object atomicity
  - ◆ Automate contention, space mgt, 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
  - ◆ Multi-object atomicity guarantees are missing or limited
    - ◆ Best bet: Support under composition constraints

# 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 & invariants
  - ◆ 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

# 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: **vars x,y initially 0** → **events x, y unseen**
    - ◆ **Activity A: send x = 1;** // (multicast send)
    - ◆ **Activity B: send y = 1;**
    - ◆ **Activity C: receive x; receive y;** // see x=1, y=0
    - ◆ **Activity D: receive y; receive x;** // 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 put/get(k)
- ◆ Doesn't extend at all under failures
  - ◆ Even for fault-tolerant systems (CAP theorem)

# Documenting Consistency Properties

## 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

# Determinism á 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



# Usability of Abstractions

Users like and use some API styles more than others

◆ **Futures: `r = ex.submit(func); ... ; use(r.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 = new CC(1, t1); ... t2.fork(); ...`**

◆ Idea: arrange to trigger an action when other(s) complete

◆ Atomic triggers for continuations avoid cascaded blocking

◆ Simple *promise*- like forms are pleasant

◆ CompletableFuture `cf = ...; cf.thenApply(f).thenApply(g)...`

◆ Non-block-structured forms messy (j.u.c CountedCompleter)

# Practical Pitfalls of Layering

## Minimal support for building libraries...

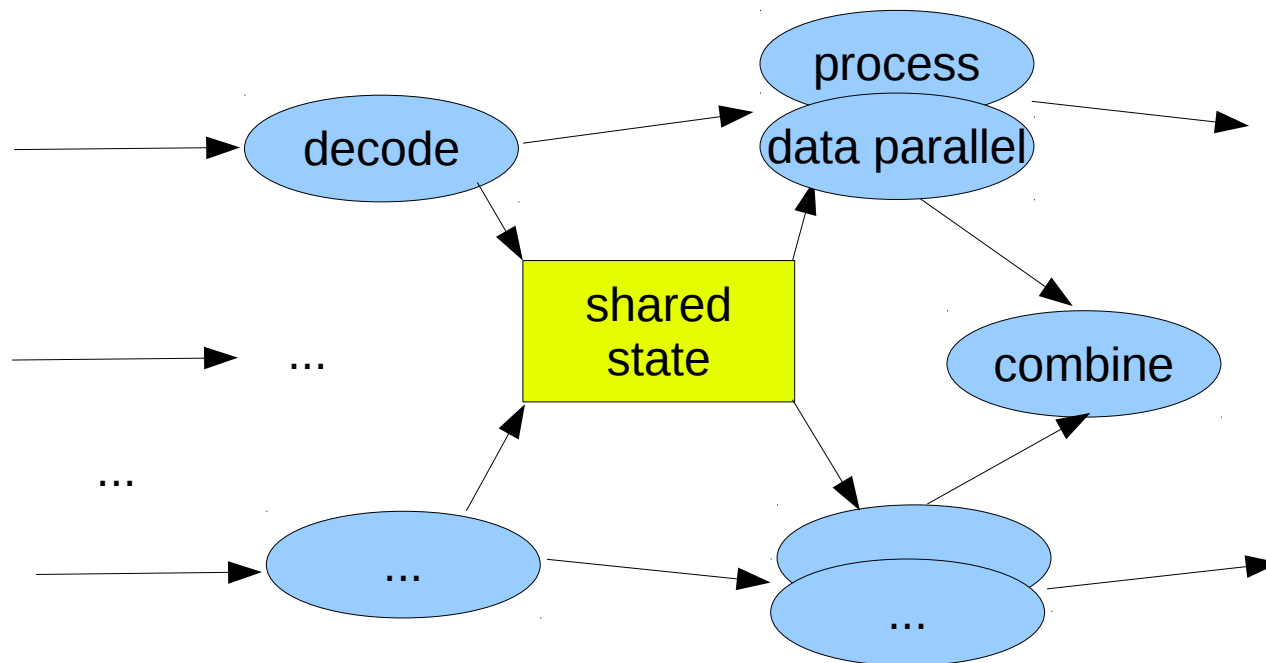
- ◆ load/store ordering, atomics, start/block/unblock threads, ...

## ... doesn't always mean easy or pleasant support:

- ◆ Coping with *Idiot Savant* dynamic compilation/optimization
  - ◆ Manual dataflow optimization
  - ◆ Using intrinsics (pseudo-bytecodes)
- ◆ Interactions with VM bookkeeping and services
  - ◆ Coping with code between the lines (e.g., safepoints)
  - ◆ Coping with GC anomalies (e.g., floating garbage)
  - ◆ Indirectly influencing memory locality, memory contention
- ◆ Coping with processor, VM, OS, Hypervisor quirks/bugs
  - ◆ Avoiding fall-off-cliff costs (e.g., when blocking threads)
- ◆ And more. For some gory details, see SPAA 2012 talk

# Latency in Concurrent Systems

- ◆ Typical system: many mostly-independent inputs; a mix of streaming and stateful processing
- ◆ QoS goals similar to RT systems
  - ◆ Minimize drops and long latency tails
  - ◆ But less willing to trade off throughput and overhead



# Software Process

- ◆ **Incrementalism requires shorter cycles**
  - ◆ Releasing a component easier than language or hardware
  - ◆ 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

# Summary: Themes revisited

- ◆ **Effective parallel programming is too diverse to be constrained by language-based policies**
  - ◆ **But often useful to layer policy over mechanism**
    - ◆ **For sake of learnability, static analysis, debugging, optimization**
- ◆ **Engineering tradeoffs lead to medium-grained abstractions**
  - ◆ **Incorporate ideas from many modes/styles**
  - ◆ **Common language design concerns are among tradeoffs**
    - ◆ **Orthogonality, generality, specification rigor, usability**
  - ◆ **May lead to multiple co-existing solutions**
- ◆ **Need diverse language support for expressing and composing them**
  - ◆ **Too few component developers can now cope**