Engineering Fine-Grained Parallelism Support for Java 7

Doug Lea
SUNY Oswego
dl@cs.oswego.edu

Prelude: why researchers write libraries

- Potentially rapid and wide adoption
  - Trying new library easier than new language
- Help developers improve quality, productivity, performance
  - Support common design patterns and use cases
    - Encapsulate otherwise difficult, tedious, slow or error-prone functionality
  - Mix of API design, algorithm design
    - Minimize compromises among efficiency, generality, ease of learning, ease of use
- Continuous evaluation
  - Developer feedback on functionality, usability, bugs
  - Ongoing software engineering, quality assurance
- Explore edges among compilers, runtimes, applications
  - Can be messy, hacky

Java Concurrency Support

- Java1.0-1.4
  - `synchronized` keyword for locking
  - `volatile` modifier for consistent access
  - `Thread` class with methods `start`, `sleep`, `interrupt`, etc
  - `Object` class monitor methods: `wait`, `notify`, `notifyAll`
- Java5-6 (JSR166)
  - Mainly improve support for “server side” programs
  - `Executors` (thread pools etc), `Futures`
  - Concurrent collections (maps, sets, queues)
  - Flexible sync (atomics, latches, barriers, RW locks, etc)
- Java7 (JSR166 “maintenance”)
  - Main focus on exploiting multi{core,proc}
  - Task-based parallelism (ForkJoin* classes)
  - Plus related fine-grained sync support

http://gee.cs.oswego.edu

Executor Pattern and API

A GOF-ish pattern with a single-method interface

```
interface Executor { void execute(Runnable w); }
```

- Separate work from workers (what vs how)
  - `ex.execute(work)`, not `new Thread(..).start()`
  - The "work" is a passive closure-like action object
- Executors implement execution policies
  - Might not apply well if execution policy depends on action
  - Can lose context, locality, dependency information
- Reduces active worker objects to very simple forms
  - Base interface allows trivial implementations like `work.run()` or `new Thread(work).start()`
  - Normally use group of threads: ExecutorService

http://gee.cs.oswego.edu

ExecutorServices

Groups of thread objects each running some variant of:
while (...) { get work and run it; }

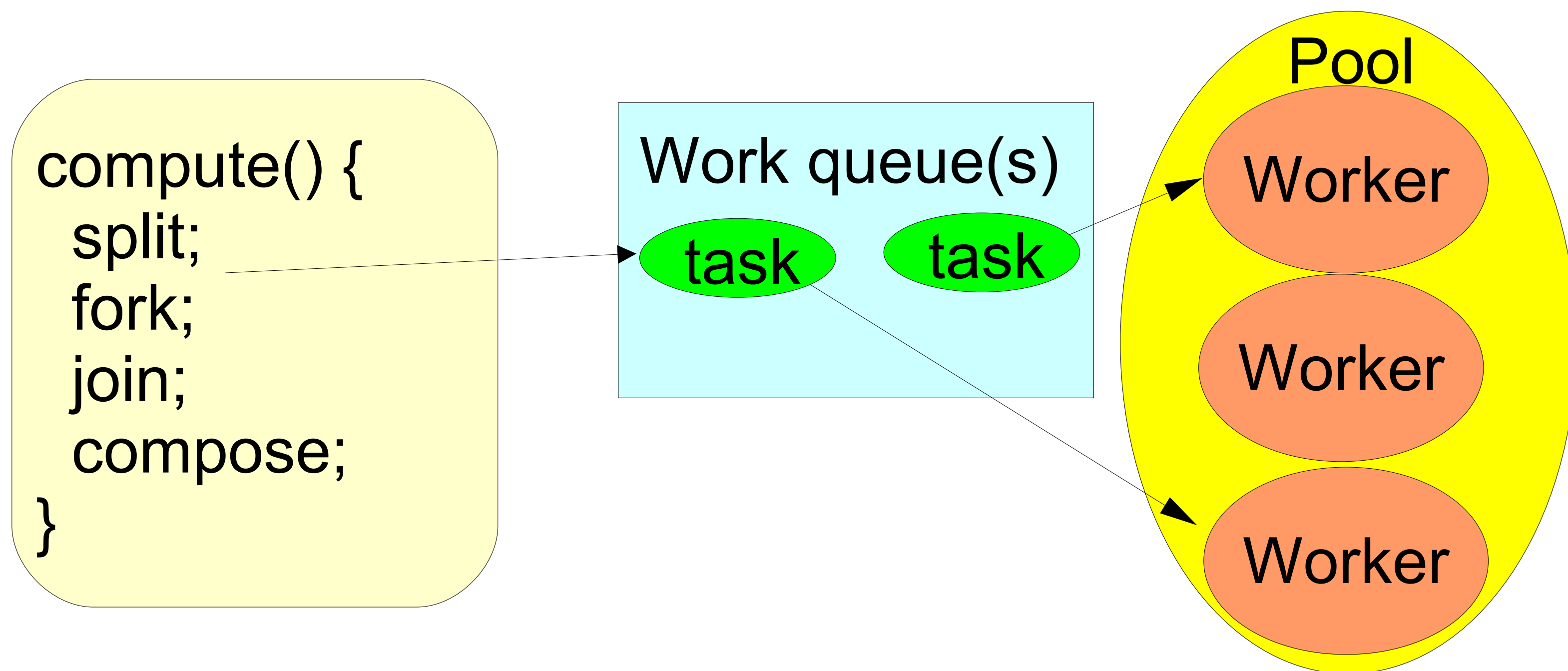
```
interface ExecutorService extends Executor {
  void shutdown();
  boolean awaitTermination(...);
  // ...
  <T> Future<T> submit(Callable<T> op);
}
```

- Can multiplex many actions on few threads
  - Can serve as entry point for a set of computations
  - Clients can wait for and extract task results via Futures
- Have lifecycles, configurations and policies
  - Task queues (FIFO, prioritized, etc), time-delayed execution, saturation handlers, ...

http://gee.cs.oswego.edu

Task-based Parallelism

- **Program splits computations into tasks**
- **Worker threads continually execute tasks**
- **Plain Executors can express "join" dependencies only indirectly via Futures**



http://gee.cs.oswego.edu

Parallel Recursive Decomposition

Typical algorithm

```
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 ForkJoin framework, must convert method to task object

ForkJoinTasks

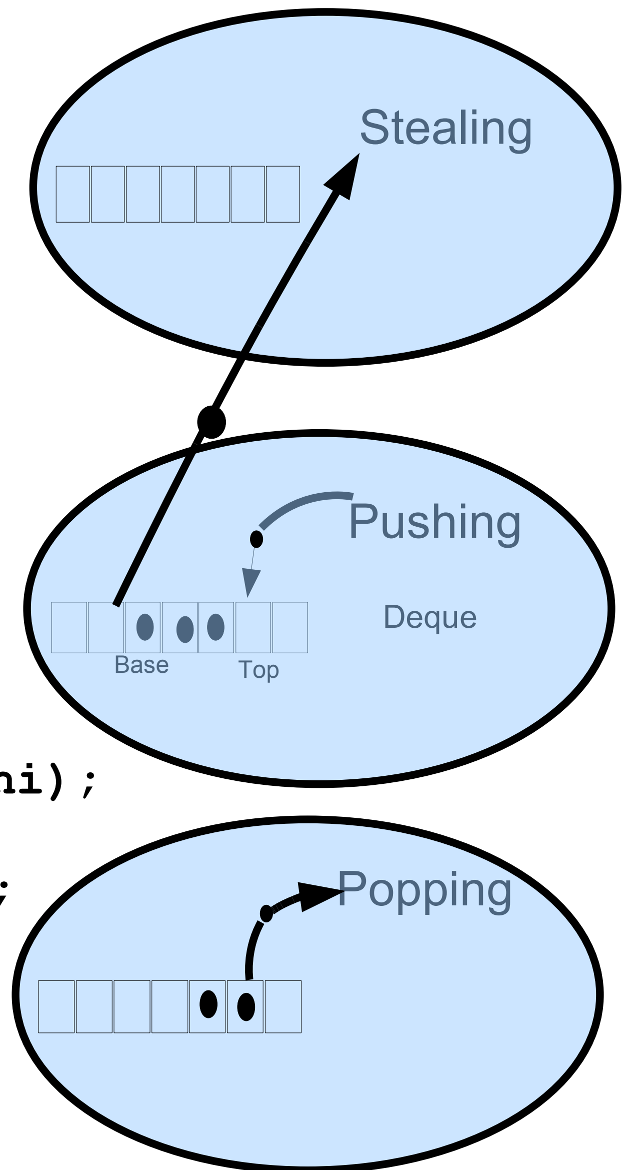
- Context-aware (unlike in plain Executor)
  - Programmers can express simple execution dependencies
- But only perform a single action (method `compute`)
  - A form of Future: Functions as Task Objects
  - A sweet spot between SIMD and unstructured async
  - Mainly used in a style pioneered by Cilk (PLDI 98)
- `fork` makes task available for execution
  - Worker threads try to steal and execute
- `join` awaits completion of `compute`
  - Often, the calling thread itself executes it
- A few other methods: cancellation, status checks, ...
  - Including support for event-like async, usable for Actors
- Subclasses RecursiveAction, RecursiveTask serve as bases for action-like vs function-like user classes

ForkJoin Sort Example

```
class SortTask extends RecursiveAction {
  final long[] array;
  final int lo; final int hi;

  SortTask(long[] array, int lo, int hi) {
    this.array = array;
    this.lo = lo; this.hi = hi;
  }

  protected void compute() {
    if (hi - lo < THRESHOLD)
      sequentiallySort(array, lo, hi);
    else {
      int m = (lo + hi) >>> 1;
      SortTask r = new SortTask(array, m, hi);
      r.fork();
      new SortTask(array, lo, m).compute();
      r.join();
      merge(array, lo, hi);
    }
  }
    // …
}
```



http://gee.cs.oswego.edu

ForkJoinPool

- Most ForkJoinTask operations are actually performed by worker threads
  - Example: fork is just `currentThread.pushTask(this)`
- Worker threads are managed by custom Executor

```
class ForkJoinPool extends AbstractExecutorService {
 <T> T invoke(ForkJoinTask<T> task);
 // ...
}
```

- Clients submit top-level tasks to the pool
  - Non-FJ clients don't have work-stealing queues etc
  - Most efficient if top-level generate many subtasks
  - Submitted tasks held in LinkedTransferQueue (see later)
- Clients can check status, await and extract results

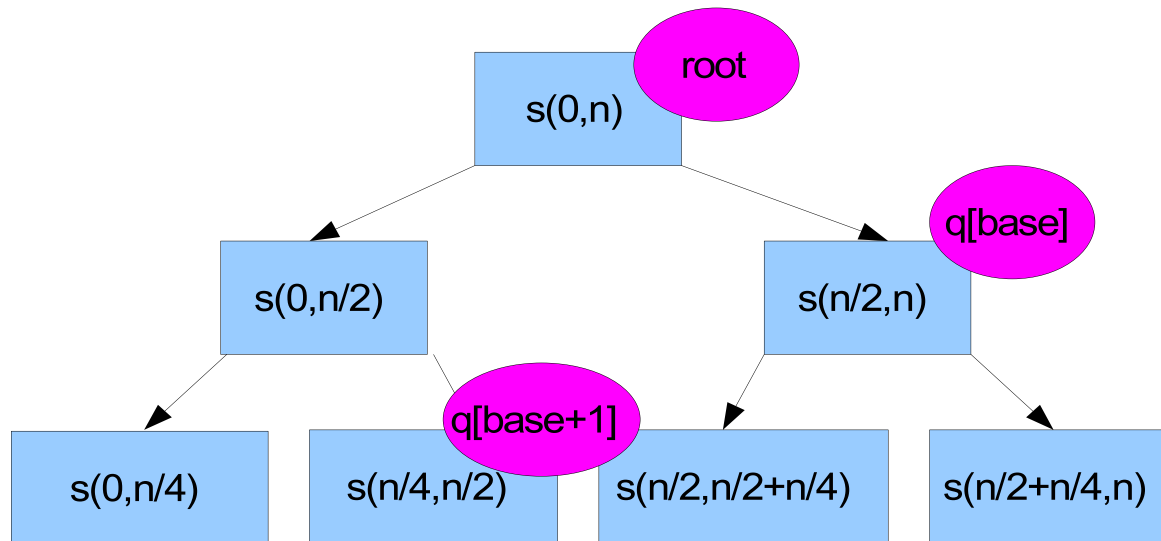
Work Stealing

- Each worker maintains own queue (actually a deque)
  - Workers steal tasks from others when otherwise idle
  - Portable: works for any number of processors
- Low overhead
  - Min: 1 int per-task space overhead, 1 atomic op per exec
  - Relies on high-throughput allocation and GC
    - Most tasks are not stolen, so task objects die unused
- 15+ years of experience (most notably in Cilk)
- But not a silver bullet – need to overcome or avoid ...
  - Basic versions don't maintain processor memory affinities
  - Task propagation delays can hurt for looping constructions
  - Overly fine granularities can hit overhead wall
  - Restricted user sync: especially, no blocking IO
  - Sizing/Scaling issues past a few hundred processors

Computation Trees and Deques

- **For recursive decomposition, deques arrange tasks with the most work to be stolen first. (See Blelloch et al for alternatives)**

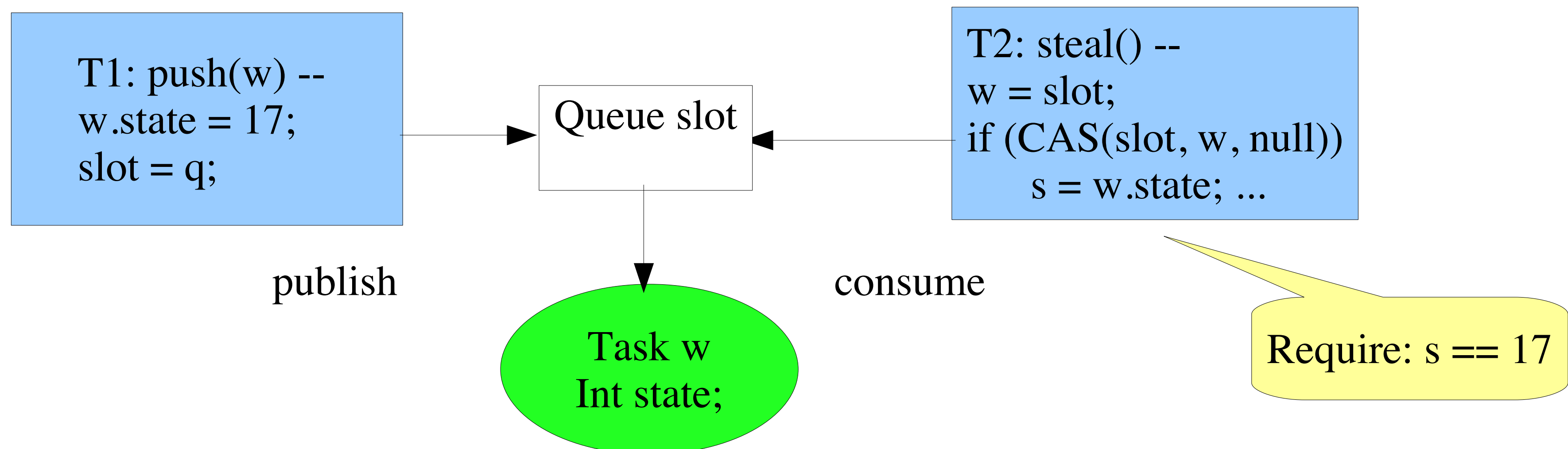
Example of method s operating on array elements 0 ... n:



http://gee.cs.oswego.edu

Transferring Tasks

- Queues perform a form of 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
- Java doesn't provide source-level map to efficient forms
  - So implementation uses JVM intrinsics



http://gee.cs.oswego.edu

Task Deque Algorithms

- Deque operations (esp push, pop) must be very fast/simple
  - Competitive with procedure call stack push/pop
- Current algorithm requires one atomic op per push+{pop/steal}
  - This is minimal unless allow duplicate execs or arbitrary postponement (See Maged Michael et al PPoPP 09)
  - Less than 5X cost for empty fork+join vs empty method call
- Uses (resizable, circular) `array` with `base` and `sp` indices
- Essentially (omitting emptiness, bounds checks, masking etc):
  - Push(t): `s = sp++; storeFence; array[s] = t;`
  - Pop(t): `if (CAS(array[sp-1], t, null)) --sp;`
  - Steal(t): `if (CAS(array[base], t, null)) ++base;`
- NOT strictly non-blocking but probabilistically so
  - A stalled ++base precludes other steals
  - But if so, stealers try elsewhere (use randomized selection)

Sample code

A variant of classic array push: q[sp++] = t (and not much slower)

Non-public method of ForkJoinWorkerThread

```
void pushTask(ForkJoinTask<?> t) {
  ForkJoinTask<?>[] q = queue;
  int mask = q.length - 1;
  int s = sp++;
  orderedPut(q, s & mask, t);
  if ((s -= base) == 0)
    pool.signalWork();
  else if (s == mask)
    growQueue();
}
```

Per-thread array-based queue with power of 2 length

inc before slot write OK

Publish via JVM intrinsic ensuring previous writes commit before slot write (inlined in the actual code)

Stealers use compareAndSet of this slot from non-null to null to privatize.

If queue was empty, wake up others using scalable event queue

Resize if full

Version 3. V1 used in JavaGrande 00, triaged out of initial JSR166

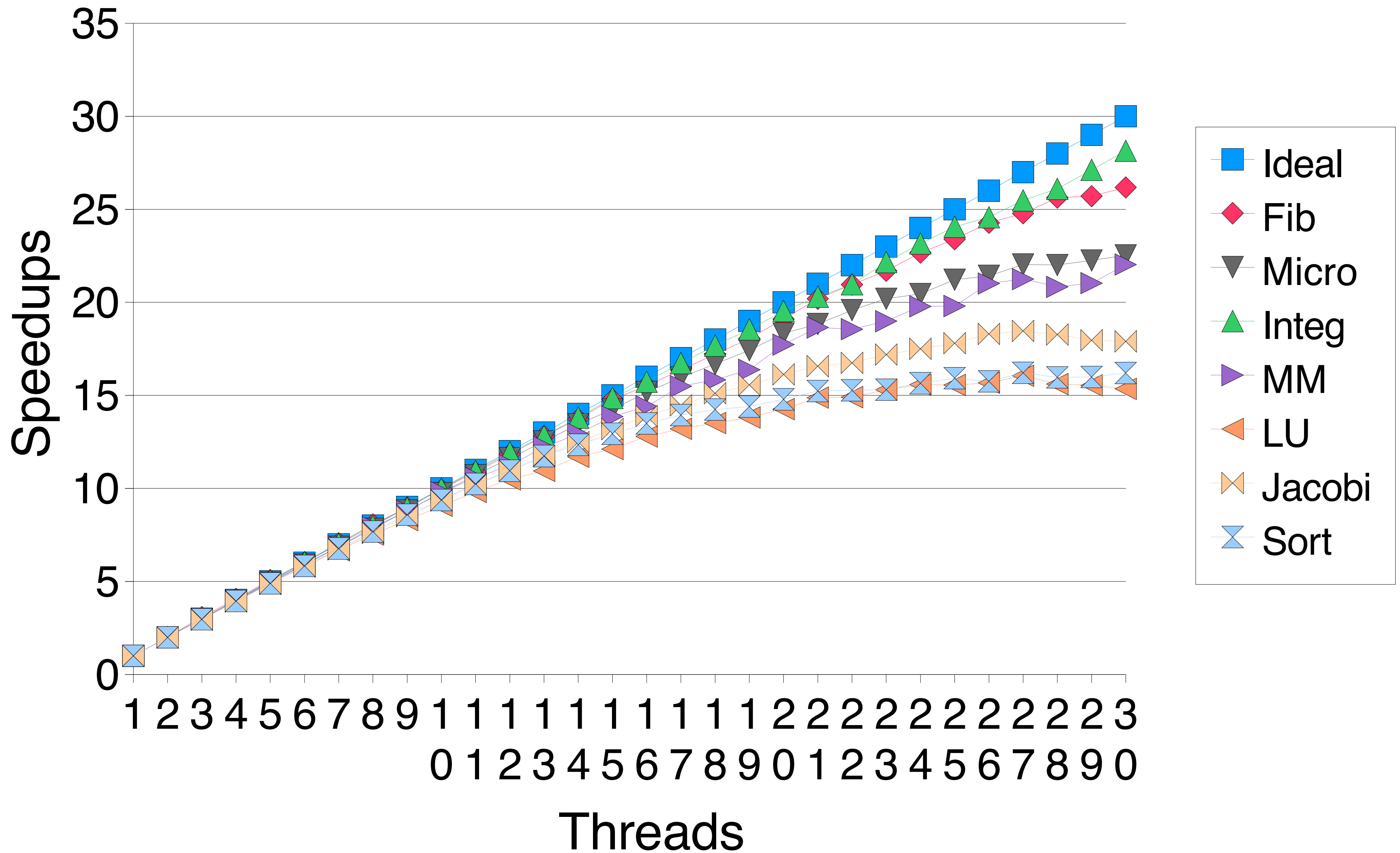
http://gee.cs.oswego.edu

Joining Tasks

- Cannot just block worker W awaiting completion of stolen task X
    - Would reduce parallelism level, maybe starve computation
- ForkJoinTask.join() interleaves two techniques, neither of which always works well alone, but together likely more effective than alternatives (such as native continuation support):
- Helping: Find/run a task that would have been on W's deque if X had not been stolen; i.e., find descendent and its tasks or stealer
    - Improves memory locality and reduces switching
    - Racy – e.g., can't find tasks if stealer is stalled
    - Requires artificial bounds to avoid search cycles
- Compensating: Create/resume spare to run on W's behalf, block W
    - Spare can finish while W unblocks
    - Spare suspends when its deque empty and too many running
    - Racy – e.g., miss that spare becomes available
    - Can increase footprint and CPU load

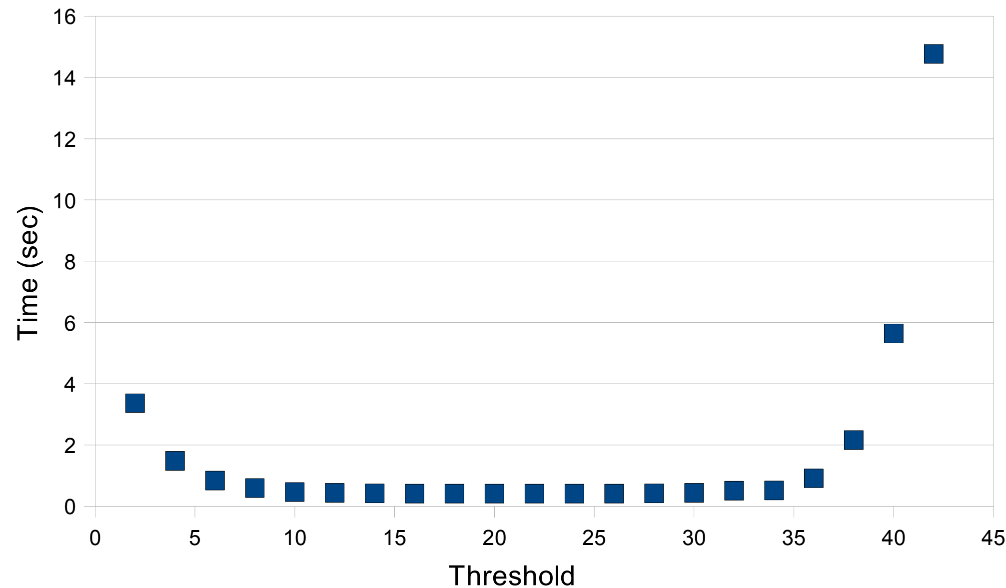
http://gee.cs.oswego.edu

Speedups on 32way Sparc



http://gee.cs.oswego.edu

Granularity Effects



Recursive Fibonacci(42) running on Niagara2

```
compute() {
  if (n <= Threshold) seqFib(n);
  else invokeAll(new Fib(n-1), new Fib(n-2)); ...}
```

- **When do you bottom out parallel decomposition?**
  - **A common initial complaint**
  - **But usually an easy empirical decision**
    - **Very shallow sensitivity curves near optimal choices**
  - **And usually easy to automate – except for problems so small that they shouldn't divide at all**

http://gee.cs.oswego.edu

Automating granularity for decomposition

- Queue-length sensing for recursive tasks
  - Each thread should help ensure progress of (idle) thieves
  - Maintain pipeline with small *constant* number of tasks available to steal in steady state, plus more on ramp up/down
    - Constant value because holds for each thread
  - Best value in part reflects overhead so not entirely analytic
    - Similar to spin lock thresholds
  - Currently use 3 plus #idleThreads:
    If (getSurplusQueuedTaskCount() > 3) seq(...) else split(...)
  - Usually identical throughput to that with manual tuning
- Can sometimes do a little better with more knowledge
  - For O(n) ops on arrays, generate #leafTasks proportional to #threads (e.g., 8 * #threads)

http://gee.cs.oswego.edu

Automating granularity for aggregation

- Example: Graph traversal
  - visit() { if (mark) for each neighbor n, fork(new Visitor(n)); }
  - Usually too few instructions to spawn task per node
- Batching based on queue sensing
  - Create tasks with node lists, not single nodes
    - Release (push) when list size exceeds threshold
  - Use batch sizes exponential in queue size (with max cap)
    - Small queue => a thread needs work, even if small batch
    - Cap protects against bad decisions during GC etc
  - Using min{128, $2^{queueSize}$} gives almost 8X speedup vs unbatched in spanning tree algorithms
    - The exact values of constants don't matter a lot
  - This approximates (in reverse) the top-down rules
- See ICPP 08 paper for details

Other Synchronization Support

- Unstructured sync not strictly disallowed but not supported
  - If one thread blocked on IO, others may spin wasting CPU
    - If *all* threads blocked on IO, pool stalls
  - ForkJoinPool provides methods to compensate for blocked tasks under external advisement
    - Basically the same compensation algorithm as in join()
- helpQuiesce(): Execute tasks until there is no work to do
  - Relies on underlying quiescence detection
    - Similar to Herlihy & Shavit section 17.6 algorithm
    - Needed anyway for pool control
  - Fastest when applicable (e.g. graph traversal)
- phaser.awaitAdvance(p): Similar to join, but triggered by phaser barrier sync
  - Based on a variant of Sarkar et al Phasers (aka clocks)

Async Actions

- Require finish() call to complete task
  - Finish of last subtask invokes parent finish
  - Replaces explicit joins with explicit continuations
    - Eliminates need for helping/compensating
  - Adds per-task linkages – more space overhead
  - Adds atomic op for each completion – slower reductions
- Base classes (*not* included) can prewire linkages and reductions

```
class Fib extends BinaryAsyncAction {
   final int n; int result;
   Fib(int n) { this.n = n; }
   public void compute() {
     if (n > T) linkAndForkSubtasks(new Fib(n-1), new Fib(n-2));
     else { result = seqFib(n); finish(); }
   }
   public void onFinish(BinaryAsyncAction x,
                        BinaryAsyncAction y) {
    result = ((Fib)x).result + ((Fib)y).result;
   }
}
```

http://gee.cs.oswego.edu

Resource Management for Tasks

- Each task is a new, usually short-lived, object
  - Each task not much more than a heap-allocated stack frame
  - Many millions of object creations per second
  - Must quickly forget them to enable GC
    - Guides many details of the algorithms
    - Coexist nicely with work-stealing parallel GC
- Some pool management must be centralized
  - Signaling work, activating spares, etc, for a mostly-unchanging pool of workers
  - Internal structures can encounter high contention
  - Needs scalable synchronization

Contention in Shared Data Structures

Mostly-Write

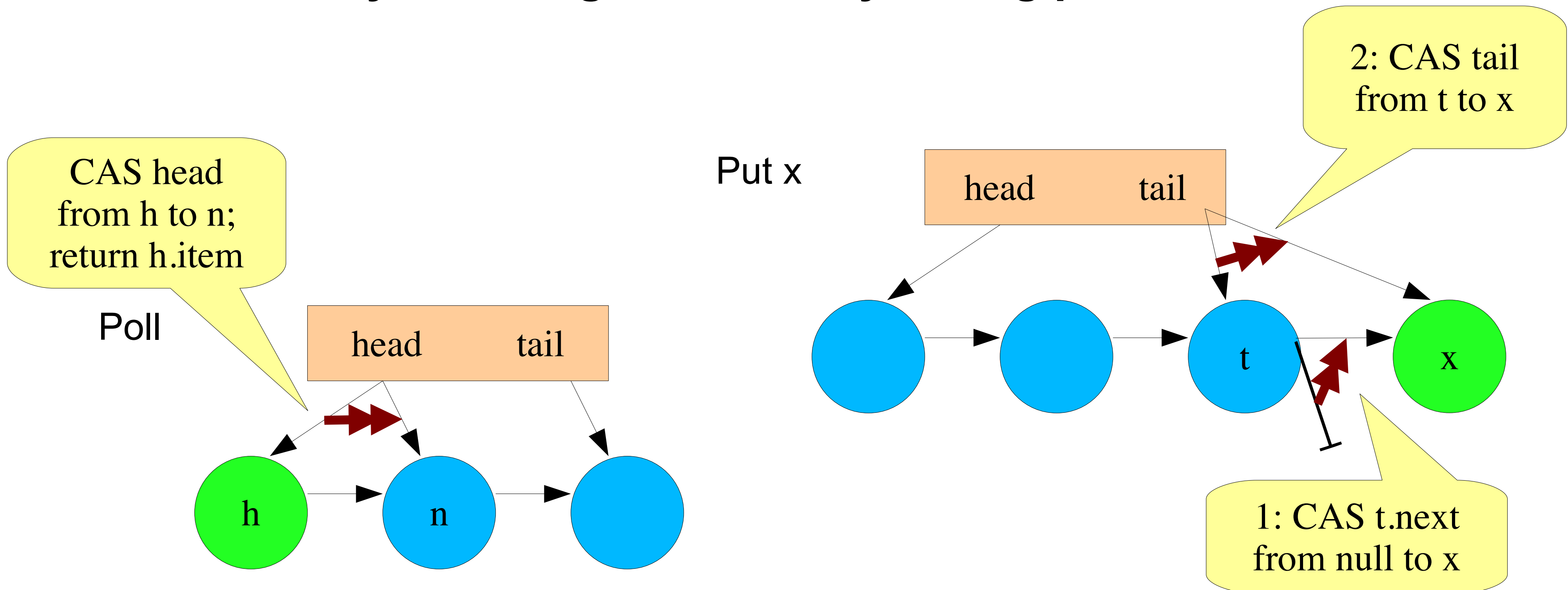
- Most producer-consumer exchanges
  - Especially queues
- Apply combinations of a small set of ideas
  - Use non-blocking sync via compareAndSet (CAS)
  - Reduce point-wise contention
  - Arrange that threads help each other make progress

Mostly-Read

- Most Maps & Sets
  - Empirically, 85% Java Map calls read-only
- 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

ConcurrentLinkedQueue

- Illustrates mostly-write techniques
- Variant of Michael & Scott Queue (PODC 1996)
  - Use retryable CAS (not lock)
  - 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



http://gee.cs.oswego.edu

Enhanced Queues

Bug report: garbage retention in ConcurrentLinkedQueue

- **Until an (unreferenced) removed node is actually GCed, its successors stay live**
  - **“floating” garbage**
- **Fixing led to idea of *slack* (Martin Buchholz):**
- **Mark removed nodes so other threads can detect staleness, help unsplice and forget successors**

RFE: Allow callers to block if empty, and block on a put awaiting a take

- **That is, support extended form of BlockingQueue**
- **Extend Scott & Scherer Dual Queues (DISC 2004)**
- **Nodes may represent either data or requests**
- **LinkedTransferQueue uses slack + blocking**
- **Elegant ideas, messy code interlacing them all**

Performance Model Oddities

- j.u.c code needs managed runtime/VM
  - Generational GC, memory model conformance, etc
  - But cope with nonuniform *idiot savant* performance
- Examples of coding between the lines
  - Presize work-stealing queue array to be much bigger than otherwise needed, to reduce cardmark contention
  - Convince JIT that signalWork (and others) ought to be compiled (vs interpreted) to minimize wakeup lags
    - Compilation counters don't reflect Amdahl's law
  - Devise portable adaptive spins to reduce expensive block/unblock
  - Work around javac adding casts for generics in identity-based compareAndSet (useless, widens race window)
  - Help implement faster hotspot/openjdk6 orderedPut support
  - Evade checked exception rules to relay task exceptions

Usage patterns, idioms, and hacks

Example: Left-spines – reuse task node down and up

```
final class SumSquares extends RecursiveAction {
   final double[] array; final int lo, hi; double result;
   SumSquares next; // keeps track of right-hand-side tasks
   SumSquares(double[] array, int lo, int hi, SumSquares next) {
     this.array = array; this.lo = lo; this.hi = hi; this.next = next;
   }
   protected void compute() {
    int l = lo; int h = hi; SumSquares right = null;
    while (h - l > 1 && getSurplusQueuedTaskCount() <= 3) {
      int mid = (l + h) >>> 1;
      (right = new SumSquares(array, mid, h, right)).fork();
      h = mid;
    }
    double sum = atLeaf(l, h);
    while (right != null && right.tryUnfork()) {
      sum += right.atLeaf(r.lo, r.hi); // Unstolen -- avoid virtual dispatch
      right = right.next;
    }
    while (right != null) {  // join remaining right-hand sides
      right.join();
      sum += right.result;
      right = right.next;
    }
    result = sum;
  }
  private double atLeaf(int l, int r) {
    double sum = 0;
    for (int i = l; i < h; ++i) // perform leftmost base step
      sum += array[i] * array[i];
    return sum;
  } }
```

http://gee.cs.oswego.edu

VM Support Issues

- Explicit memory fences and more complete atomics
  - Fences API, proposed but unloved
- Tail-recursion
  - Needed internally to loopify recursion that entails callbacks
- Boxing
  - Must avoid arrays of boxed elements
- Guided inlining / macro expansion ?
  - Avoid megamorphic compute methods at leaf calls
- Native Continuations?
  - Not clear they'd ever be better/faster than alternatives
- VM internals
  - Reducing GC-based contention (e.g., on cardmarks)
  - Counters cannot guide compilation of sequential bottlenecks
  - Allowing idle threads help with GC (maybe via Thread.yield)

Processor/Platform Support Issues

- Fast, simple fences for fast, simple ownership transfer
  - Needed in basic work-stealing operations
  - Fast enough not to need to avoid on recent X86 and Sparc
  - Generating fast code on non-TSO too hard for JIT?
    - Would sometimes need heavy escape analysis
- More cache-aware dynamic thread-to-core/processor mappings
  - Balancing sharing/pollution within, vs contention between
  - Most non-HPC applications cannot statically determine
- Someday: integration with GPUs and SIMD
  - Use for some operations on aggregates
- Someday: Portable simple low-level transactions
  - Multiple-location LL/SC or minor variants
- Continuing needs for java.util.concurrent:
  - Lower overhead threads, blocking/unblocking, time/timing

Libraries and Language Evolution

- Automate parallel recursive divide and conquer across combined operations on aggregates

```
class Student { String name; int gradYear; double gpa; }

ParallelArray<Student> students = ParallelArray.create(...);

double highestGpa = students.withFilter(graduatesThisYear)
                            .withMapping(selectGpa)
                            .max();
```

- Too ugly without syntax support for closures

```
Ops.Predicate<Student> graduatesThisYear =
new Ops.Predicate<Student>() {
  public boolean op(Student s) {
    return s.gradYear == THIS_YEAR; } };
```

Current Status

- Snapshots available in package jsr166y at: http://gee.cs.oswego.edu/dl/concurrency-interest/
  - Seems to have a few hundred early users
  - Used by Fortress, Clojure, Scala, etc runtimes
  - Used by other optional packages in Java, Groovy, etc
- Ongoing work
  - Simplifying APIs based on user experience
    - Mainly, removing some methods
  - Improving resilience to abuse (e.g., blocking on IO)
  - JDK release preparation
    - More testing, reviews, spec clarifications, tutorials, etc
- Under consideration
  - Replace FJP async mode with event/actor-friendly class(es)

Postscript: Developing Libraries

- API design is a social process
  - Single visions are good, those that pass review are better
- Specification and documentation require broad review
  - Even so, by far most submitted j.u.c bugs are spec bugs
- Release engineering requires perfectionism
  - Lots of QA: tests, reviews. Still not enough
- Standardization required for widespread use
  - JCP both a technical and political body
- Need tutorials etc written at many different levels
  - Users won't read academic papers to find out how/why to use
- Creating new components leads to new developer problems
  - Example: New bug patterns for findBugs

(Extra slides follow)

Consistency

- Processors do not intrinsically guarantee much about memory access orderings
  - Neither do most compiler optimizations
    - Except for classic data and control dependencies
  - Not a bug
    - Globally ordering all program accesses can eliminate parallelism and optimization → unhappy programmers
- Need memory model to specify guarantees and how to get them when you need them
  - Initial Java Memory Model broken
  - JSR133 overhauled specs but still needs some work

Language-Based Memory Models

- Distinguish
  - sync *accesses* (locks, volatiles, atomics) vs
  - normal accesses (reads, writes)
- Require strong ordering properties among sync
  - Usually “strong” means Sequentially Consistent
- Allow as-if-sequential reorderings among normal
  - Usually means: obey seq data/control dependencies
- Restrict reorderings between sync vs normal
  - Many rule choices, few “obviously” right or intuitive
    - And not tied to ownership semantics or separation logic
  - Special rules for cases like final fields
    - Specs based on only two flavors of access not enough

Mapping Memory Models

- Enforce language rules in hardware & optimizers
  - May require expensive CPU fences, atomic ops
- Often cheaper for structured ownership vs locks
  - A consideration in devising concurrent algorithms
  - But effects are only conditionally correct
  - Example: cheaper store-fence if never modify after publish
  - Difficult specs: must say what happens if not used correctly
- Poor language support for special-mode accesses
  - Requires intrinsics operating on addresses (not values)
  - Alternatively, source-level control over fences
  - Not very usable in Java
    - Still, I use them a lot
    - Better language design still a research problem

Consistency Issues are Inescapable

- Occur in remote message passing
  - Memory model mapping to distributed platforms expensive
  - Many groups don't need strong consistency
  - But encounter anomalies
- Example (“*IRIW*”): x,y initially 0

```
Node A: send x = 1;                 // (multicast send)
Node B: send y = 1;
Node C: receive x; receive y;   // see x=1, y=0
Node D: receive y; receive x;   // see y=1, x=0
```

- Full avoidance as expensive as full MM mapping – atomic multicast, distributed transactions
  - Moreso when must tolerate remote failure
- Occur in local messaging: Processes, Isolates, ...
  - Usually rely on implicit OS-level consistency model

http://gee.cs.oswego.edu