Supporting Fine-Grained Parallelism in Java 7

Doug Lea
SUNY Oswego
dl@cs.oswego.edu

Outline

- Background
  - Language and Library support for concurrency
- Fine-grained task-based parallelism
  - Work-stealing algorithms
  - APIs, usages, and platform support

Prelude: why researchers write libraries

- Because that's where many interesting problems are
  - Compromise as little as possible between very fast and very easy to use. Mix of API design, algorithm design, SE.
- Help developers to improve Quality, Productivity, Performance
  - When component functionality is {Common, Difficult, Tedious, Error-Prone} then put it in a library
  - When programmers are seen to have trouble structuring code, invent new abstractions that make it easier
  - When obvious implementations are slow, put faster ones in library
- Coexists with goal of making constructions easier
  - New languages, platforms, computing models
  - Improve usability of existing languages and platforms

Java Concurrency Support

- **Java 1.0-1.4**
  - **Threads, locks, monitors**
- **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}**
  - **A substrate for Fortress, X10, Scala, etc**
  - **Task-based parallelism (forkjoin package)**
  - **Plus better fine-grained sync for Thread-based programs**

Core Java Concurrency Support

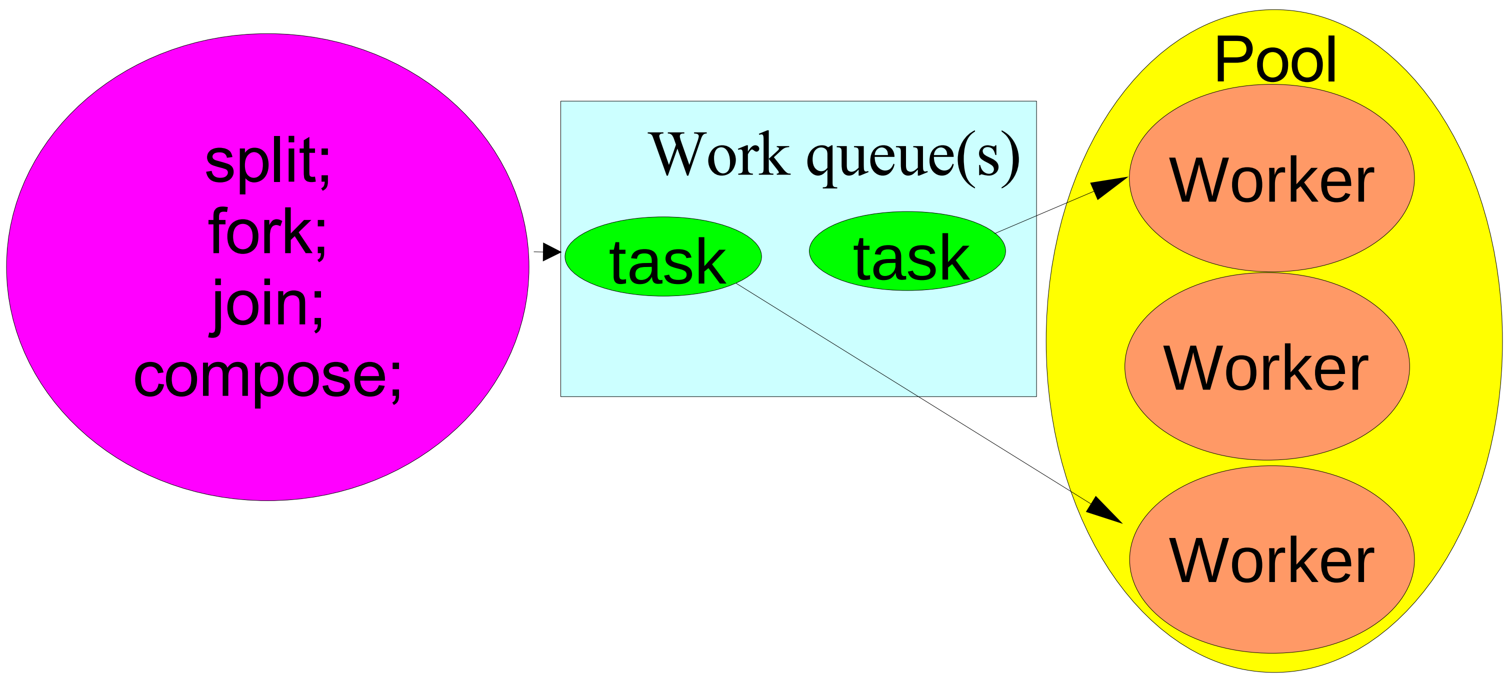
- **Built-in language features:**
  - `synchronized` **keyword**
    - **"monitors" part of the object model**
  - `volatile` **modifier**
    - **Roughly, reads and writes act as if in synchronized blocks**
- **Core library support:**
  - `Thread` **class methods**
    - `start, sleep, yield, isAlive, getID, interrupt, isInterrupted, interrupted, ...`
  - `Object` **methods:**
    - `wait`, `notify`, `notifyAll`

java.util.concurrent

- Executor framework
  - ThreadPools, Futures, CompletionService
- Atomic variables (subpackage `java.util.concurrent.atomic`)
  - JVM support for compareAndSet operations
- Lock framework (subpackage `java.util.concurrent.locks`)
  - Including Conditions & ReadWriteLocks
- Concurrent collections
  - Queues, Lists, Sets, Maps
- Synchronizers
  - Semaphores, Barriers, Exchangers, CountDownLatches

Task-based Parallelism

- Program splits computations into tasks
- Worker threads continually execute tasks
- Plain form is basis for existing j.u.c Executor framework



Work-Stealing

- Scalable version of Executor (in new “forkjoin” package)
- Eliminates most global synchronization
  - Each worker maintains own queue (actually a Deque)
  - Workers steal tasks from others when otherwise idle
  - Still maintain central “submission queue” and other mgt
- Minimizes per-task creation and bookkeeping overhead
  - Only one `int` of per-task space overhead
  - Relies on high-throughput allocation and GC
    - Most tasks are not stolen, so task objects die unused
- Minimizes per-task synchronization
  - But restricts the kinds of sync allowed
    - Mainly joining (awaiting completion) of subtasks

Parallel Recursive Decomposition

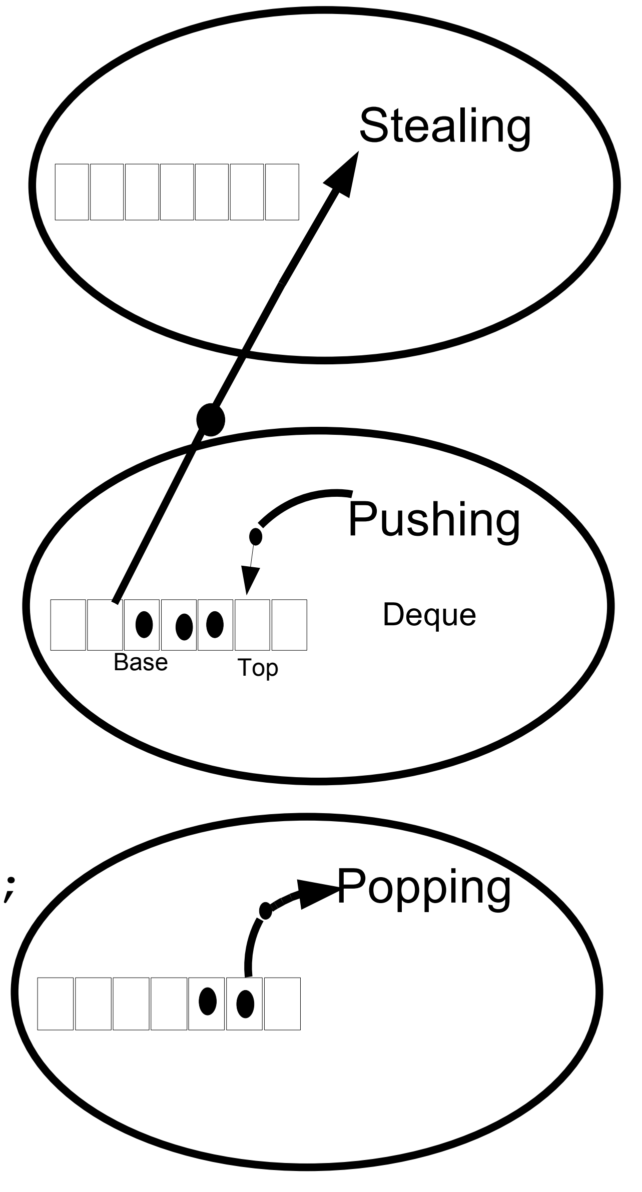
- Typical algorithm

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

- To use framework, must convert method to task object
- Under work-stealing, the algorithm itself drives the scheduling
- Many variants and extensions, but this simple form is usually best behaved and widely applicable

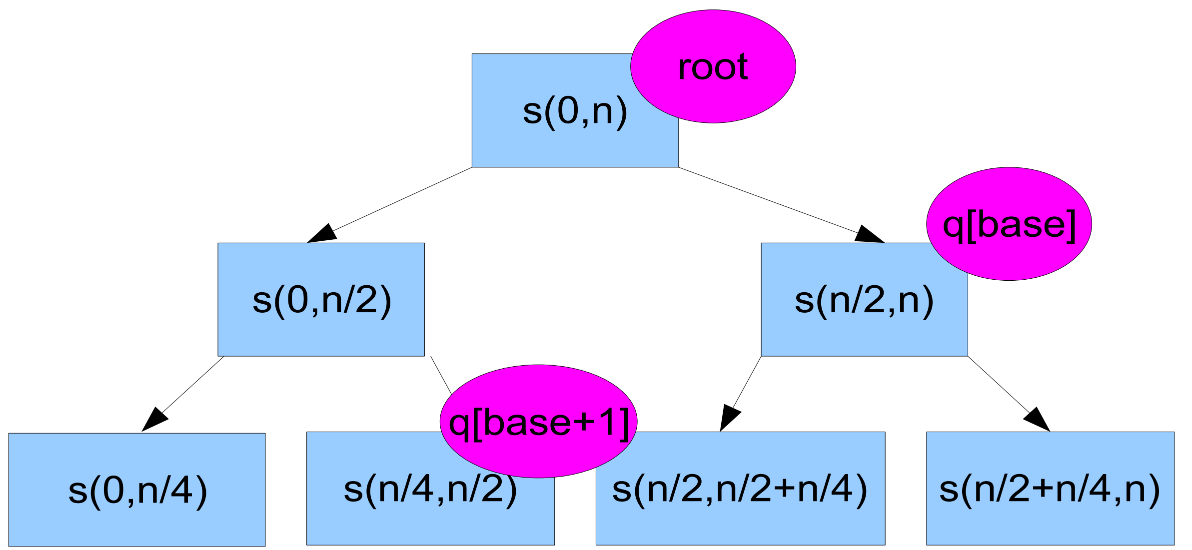
Fork/Join Sort Example

```
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;
        forkJoin(new SortTask(array, lo, m),
                 new SortTask(array, m, hi));
        merge(array, lo, hi);
      }
    }
    // ...
}
```

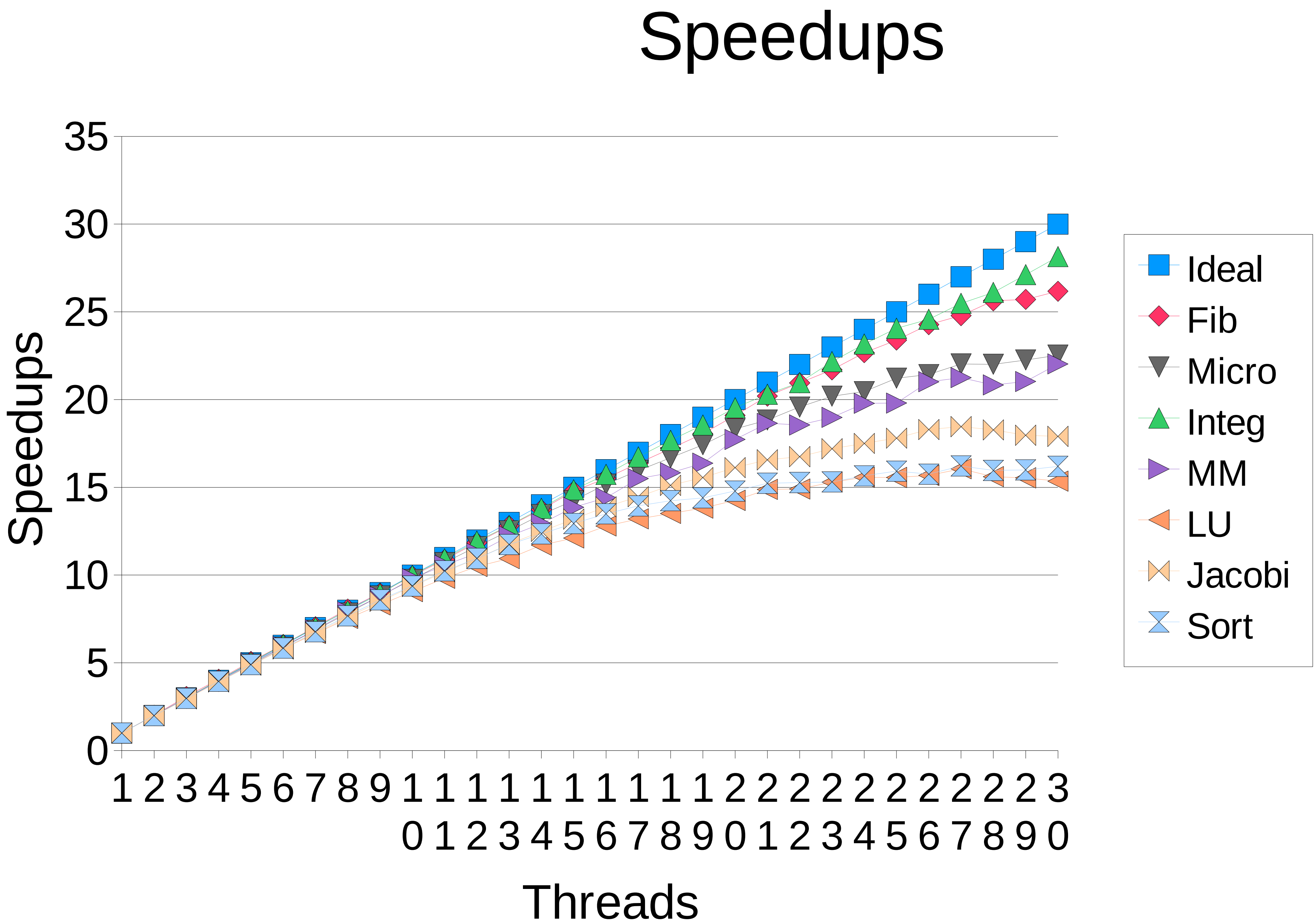


Computation Trees and Deques

- 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
  - Where forkJoin(a, b) => push(a); exec(b); join(a)
    - (Alternatives discussed later)



Speedups on 32way Sparc



Why Work-Stealing

- Portable scalability
  - Programs work well with any number of processors/cores
- Load-balancing
  - Keeps processors busy, improves throughput
- Robustness
  - Can afford to use small tasks (as few as 100 instructions)
- 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 big overhead wall
  - Restricted sync restricts range of applicability
  - Sizing/Scaling issues past a few hundred processors

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)
  - Approx 5X cost for empty forkjoin vs empty method call
- Uses (resizable, circular) `array` with `base` and `sp` indices
- Essentially (omitting emptiness, bounds checks, masking etc):
  - Push(t): `storeFence; array[sp] = t; ++sp;`
  - 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)

Synchronization Support

- Must support diverse but structured coordination techniques
  - Support multiple techniques so only pay for what you need
  - Can also rely on j.u.c. nonblocking collections etc
- Unstructured sync not strictly disallowed but not supported
  - If one thread blocked on IO, others may spin wasting CPU
- 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)
- Joining (see next)

Joining

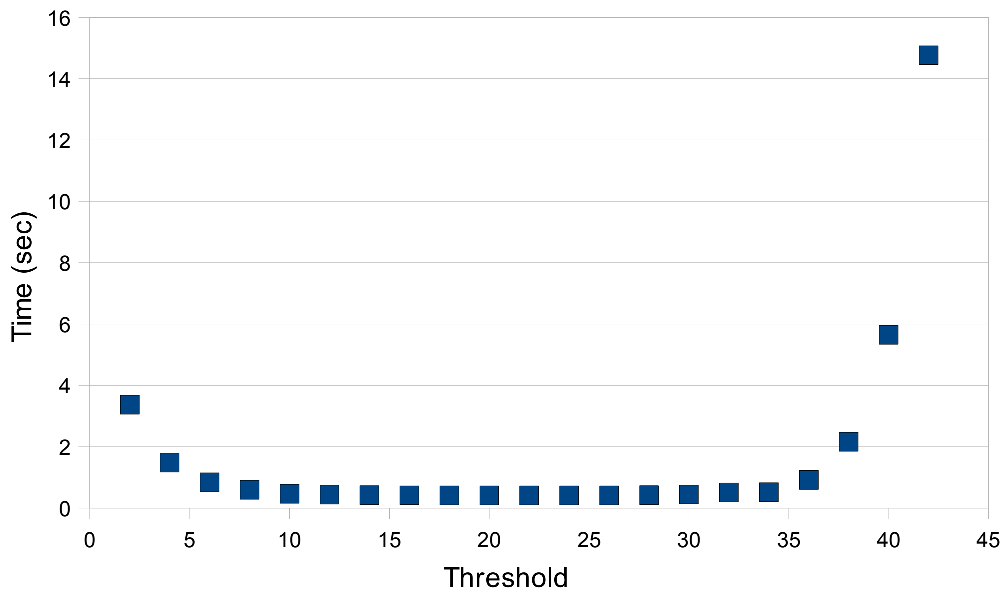
- Three supported techniques for dependence on task t that was stolen (or never owned) but not yet done:
- t.helpJoin()
  - Busy-help by stealing and running other tasks until t done
    - No atomics, blocking, or signals
  - Usually fast but only works for tree-structured computations
    - Otherwise a continuation can become permanently buried
- t.join()
  - Block thread, enable a spare to steal/exec tasks
  - When t done, wake up, let spare suspend when next idle
  - More overhead but maintains parallelism without lockup
    - Spare threads emulate continuations
- Using AsyncActions (see next)

Async Actions

- Require finish() call to complete task
  - Finish of last subtask invokes parent finish
  - Replaces explicit joins with explicit continuations
  - Adds per-task linkages – more space overhead
  - Adds atomic op for each completion – slower reductions
- Subclasses (Binary, Linked) 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;
   }
}
```

Granularity Effects



- Recursive Fibonacci(42) running on Niagara2

```
compute() {
  if (n <= Threshold) seqFib(n);
  else forkJoin(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 just as easy to automate

Automating granularity for decomposition

- Based on 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
    - But holds framework-wide, not per program
    - Similar to e.g. 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)

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
    - As usual, 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 Support

- Additional flavors of ForkJoinTasks
  - Recursive, Async, Phased
    - Result-full Tasks and result-less Actions
    - Phased (upcoming) reduces re-spawn costs in loops
- Direct ForkJoinWorkerThread access
  - Exposes push, pop etc to allow better tuning
  - Subclassable – can add per-thread state etc
- Common utilities
  - Example: Per-worker-thread random number generator
- Management and Monitoring
  - Submission queues, Shutdown, pool resizing
  - Track active threads, steals, etc

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 -- invoke compute to avoid virtual dispatch
      right = right.next;
    }
    while (right != null) {  // join remaining right-hand sides
      right.helpJoin();
      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;
  } }
```

VM Support Issues

- Explicit memory fences and more complete atomics
  - Underway (proposed Fences API)
- Allocation and high-throughput GC
  - Including issues like cardmark contention
  - Allowing idle threads help with GC (maybe via Thread.yield)
- Tail-recursion
  - Needed internally to loopify recursion including callbacks
- Boxing
  - *Must* avoid arrays of boxed elements
- Guided inlining / macro expansion
  - Avoid megamorphic compute methods at leaf calls
- Continuations?
  - Not clear they'd ever be faster than alternatives

Possible Java Library Extensions

- Support apply, select, map, scan, reduce, etc on aggregates
  - Can be done via library support, not language support
  - But function-types and closure bodies painful to express
- Example: ParallelArray

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

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

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

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

Ops.ObjectToDouble<Student> selectGpa = new Ops.ObjectToDouble<Student>() {
   public double op(Student student) { return student.gpa;} };
```

Current Status

- Snapshots available in package jsr166y at: http://gee.cs.oswego.edu/dl/concurrency-interest/index.html
  - Seems to have a few hundred early users
  - Targetting at least core functionality for Java7
  - Used by Fortress, X10, Scala, etc runtimes
- Ongoing work
  - JDK release preparation
    - More testing, reviews, spec clarifications, tutorials, etc
- Further out
  - Better integration with transactional support, thread-based and event-based parallelism

Postscript: researchers cannot do it alone

- API design is a social process
  - Single visions are good, but 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
- Developers will not read academic papers to figure out how or why to use components
  - Need tutorials etc written at many different levels
- Creating new components leads to new developer problems
  - Example: New bug patterns for findBugs