From Concurrent to Parallel

Library-based parallelism in JDK 7

Brian Goetz

Sr. Staff Engineer, Sun Microsystems

brian.goetz@sun.com

Overview

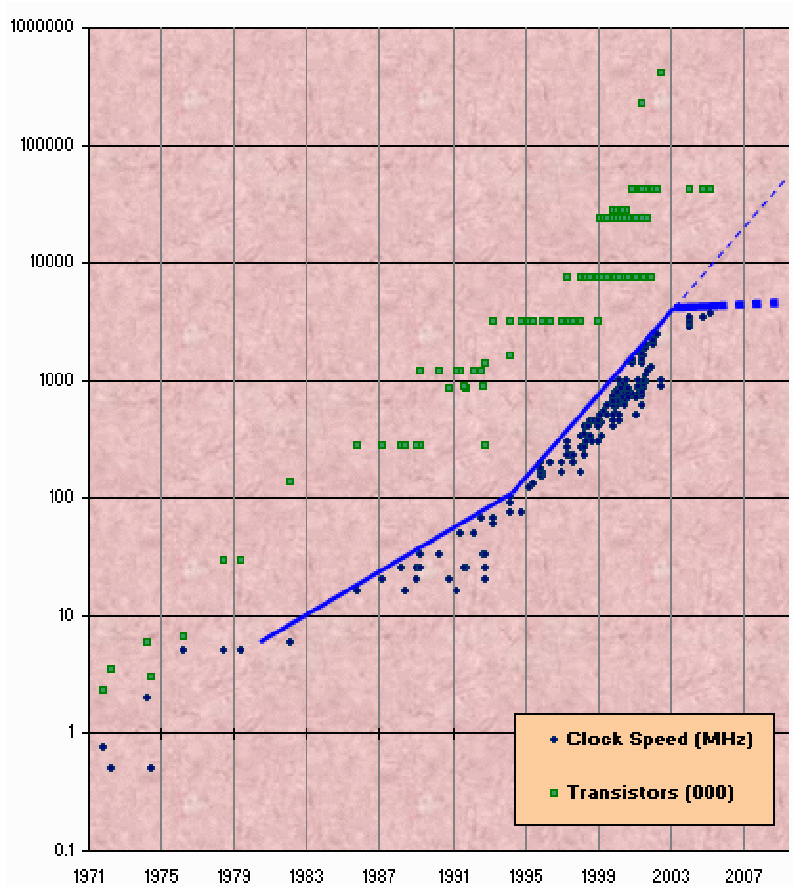
JDK 5.0 added lots of useful classes for coarse-grained concurrency.

Hardware trends now require us to look deeper in our applications for latent parallelism.

The fork-join framework in JDK 7 can help.

Hardware trends

- As of ~2003, we stopped seeing increases in CPU clock rate
- Moore's law has not been repealed!
  - Giving us more cores per chip rather than faster cores
    - Maybe even slower cores
- Chart at right shows clock speed of Intel CPU releases over time
  - Exponential increase until 2003
  - No increase since 2003
- Result: many more programmers become concurrent programmers (maybe reluctantly)



Hardware trends

- “The free lunch is over”
  - For years, we had it easy
    - Always a faster machine coming out in a few months
  - Can no longer just buy a new machine and have our program run faster
    - Even true of many so-called concurrent programs!
- Challenge #1: decomposing your application into units of work that can be executed concurrently
- Challenge #2: Continuing to meet challenge #1 as processor counts increase
  - Even so-called scalable programs often run into scaling limits just by doubling the number of available CPUs
  - Need coding techniques that parallelize efficiently across a wide range of processor counts

Hardware trends

- Primary goal of using threads has always been to achieve *better CPU utilization*
  - But those hardware guys just keep raising the bar
- In the old days – only one CPU
  - Threads were largely about *asynchrony*
    - Utilization improved by doing other work during I/O operations
- More recently – handful (or a few handfuls) of cores
  - Coarse-grained parallelism usually enough for reasonable utilization
    - Application-level requests made reasonable task boundaries
    - Thread pools were a reasonable scheduling mechanism
- Soon – all the cores you can eat
  - May not be enough concurrent user requests to keep CPUs busy
  - May need to dig deeper to find latent parallelism
  - Shared work queues become a bottleneck

Hardware trends drive software trends

- Languages, libraries, and frameworks shape how we program
  - All languages are Turing-complete, but…the programs we *actually* write reflect the idioms of the languages and frameworks we use
- Hardware shapes language, library, and framework design
  - The Java language had thread support from day 1
    - But early support was mostly useful for asynchrony, not concurrency
    - Which was just about right for the hardware of the day
- As MP systems became cheaper, platform evolved better library support for *coarse-grained* concurrency (JDK 5)
  - Principal user challenge was identifying reasonable task boundaries
- Programmers now need to exploit *fine-grained* parallelism
  - Better library support will help!
  - May be able to borrow classical parallel programming techniques
  - We need to be on the lookout for latent parallelism

Finding finer-grained parallelism

- User requests are often too coarse-grained a unit of work for keeping many-core systems busy
  - May not be enough concurrent requests
  - Possible solution: find parallelism *within* existing task boundaries
- Most promising candidate is sorting and searching
  - Amenable to parallelization
    - Sorting can be parallelized with merge sort
    - Searching can be parallelized by searching sub-regions of the data in parallel and then merging the results
  - Can improve response time by using more CPUs
    - May actually use more total CPU cycles, but less wall-clock time
    - Response time may be more important than total CPU cost
      - Human time is valuable!

Finding finer-grained parallelism

- Example: stages in the life of a database query
  - Parsing and analysis
  - Plan selection (may evaluate many candidate plans)
  - I/O (already reasonably parallelized)
  - Post-processing (filtering, sorting, aggregation)
    - `SELECT first, last FROM Names ORDER BY last, first`
    - `SELECT SUM(amount) FROM Orders`
    - ```
      SELECT student, AVG(grade) as avg FROM Tests
         GROUP BY student
         HAVING avg > 3.5
      ```
- Plan selection and post-processing phases are CPU-intensive
  - Could be sped up with more parallelism
      ```

Running example: select-max

- Simplified example: find the largest element in a list
  - O(n) problem
  - Obvious sequential solution: iterate the elements
    - For very small lists the sequential solution is obviously fine
    - For larger lists a parallel solution will clearly win
      - Though still O(n)

```
class MaxProblem {
    final int[] nums;
    final int start, end, size;

    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i=start; i<end; i++)
            max = Math.max(max, nums[i]);
        return max;
    }

    public MaxProblem subproblem(int subStart, int subEnd) {
        return new MaxProblem(nums, start+subStart, start+subEnd);
    }
}
```

First attempt: Executor+Future

- We can divide the problem into N disjoint subproblems and solve them independently
  - Then compute the maximum of the result of all the subproblems
  - Can solve the subproblems concurrently with invokeAll()

```
Collection<Callable<Integer>> tasks = ...
for (int i=0; i<N; i++)
   tasks.add(makeCallableForSubproblem(problem, N, i));
List<Future<Integer>> results = executor.invokeAll(tasks);
int max = -Integer.MAX_VALUE;
for (Future<Integer> result : results)
   max = Math.max(max, result.get());
```

First attempt: Executor+Future

- A reasonable choice of N is Runtime.availableProcessors()
  - Will prevent threads from competing with each other for CPU cycles
  - Problem is "embarassingly parallel"
- But this approach has several inherent scalability limits
  - Shared work queue in Executor eventually becomes a bottleneck
  - If some subtasks finish faster than others, may not get ideal utilization
    - Can address by using smaller subproblems
    - But this increases contention costs
- Code is clunky!
  - Subproblem extraction prone to fencepost errors
  - Find-maximum loop duplicated
- Clunky code => people won't bother with it

Parallelization technique: divide-and-conquer

- Divide-and-conquer breaks down a problem into subproblems, solves the subproblems, and combines the result
- Example: merge sort
  - Divide the data set into pieces
  - Sort the pieces
  - Merge the results
  - Result is still *O(n log n)*, but subproblems can be solved in parallel
    - Parallelizes fairly efficiently – subproblems operate on disjoint data
- Divide-and-conquer applies this process recursively
  - Until subproblems are so small that sequential solution is faster
  - Scales well – can keep many CPUs busy

Divide-and-conquer

- Divide-and-conquer algorithms take this general form

```
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return problem.solveSequentially();
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(problem.extractLeftHalf());
            right = solve(problem.extractRightHalf());
        }
        return combine(left, right);
    }
}
```

- The invoke-in-parallel step waits for both halves to complete
  - Then performs the combination step

Fork-join parallelism

- The key to implementing divide-and-conquer is the *invoke-in-parallel* operation
  - Create two or more new tasks (fork)
  - Suspend the current task until the new tasks complete (join)
- Naïve implementation creates a new thread for each task
  - Invoke Thread() constructor for the fork operation
  - Thread.join() for the join operation
  - Don’t actually want to do it this way
    - Thread creation is expensive
    - Requires O(log n) idle threads
- Of course, non-naïve implementations are possible
  - Package java.util.concurrent.forkjoin proposed for JDK 7 offers one
  - For now, download package jsr166y from `http://gee.cs.oswego.edu/dl/concurrency-interest/index.html`

Solving select-max with fork-join

- The RecursiveAction class in the fork-join framework is ideal for representing divide-and-conqure solutions

```
class MaxSolver extends RecursiveAction {
    private final MaxProblem problem;
    int result;

    protected void compute() {
        if (problem.size < THRESHOLD)
            result = problem.solveSequentially();
        else {
            int m = problem.size / 2;
            MaxSolver left, right;
            left = new MaxSolver(problem.subproblem(0, m));
            right = new MaxSolver(problem.subproblem(m,
                                  problem.size));
            forkJoin(left, right);
            result = Math.max(left.result, right.result);
        }
    }
}

ForkJoinExecutor pool = new ForkJoinPool(nThreads);
MaxSolver solver = new MaxSolver(problem);
pool.invoke(solver);
```

Fork-join example

- Example implements RecursiveAction
  - The forkJoin() method creates two new tasks and waits for them
  - ForkJoinPool is like an Executor, but optimized for fork-join tasks
    - Waiting for other pool tasks risks *thread-starvation deadlock* in standard executors
- Implementation can avoid copying elements
  - Different subproblems work on disjoint portions of the data
    - Which also happens to have good cache locality
    - Data copying would impose a significant cost
  - In this case, data is read-only for the entirety of the operation
- Other useful task base classes
  - RecursiveTask for result-bearing tasks
  - AsyncAction for tasks with asynchronous completion
  - CyclicAction for parallel iterative tasks

Performance considerations

- How low should the sequential threshold be set?
- Two competing performance forces
  - Making tasks smaller enhances parallelism
    - Increased load balancing, improves throughput
  - Making tasks larger reduces coordination overhead
    - Must create, enqueue, dequeue, execute, and wait for tasks
- Fork-join task framework is designed to minimize per-task overhead for *compute-intensive* tasks
  - The lower the task-management overhead, the lower the sequential threshold can be set
  - Traditional Executor framework works better for tasks that have a mix of CPU and I/O activity

Performance considerations

- Fork-join offers a *portable* way to express many parallel algorithms
  - Code is independent of the execution topology
  - Reasonably efficient for a wide range of CPU counts
  - Library manages the parallelism
    - Frequently no additional synchronization is required
- Still must set number of threads in fork-join pool
  - Runtime.availableProcessors() is usually the best choice
    - Larger numbers won't hurt much, smaller numbers will limit parallelism
- Must also determine a reasonable sequential threshold
  - Done by experimentation and profiling
  - Mostly a matter of avoiding "way too big" and "way too small"

Performance considerations

- Table shows speedup relative to sequential for various platforms and thresholds for 500K run (bigger is better)
  - Pool size always equals number of HW threads
    - No code differences across HW platforms
  - Can't expect perfect scaling, because framework and scheduling introduce some overhead
- Reasonable speedups for wide range of threshold

	Threshold=500k	Threshold=50K	Threshold=5K	Threshold=500	Threshold=50
Dual Xeon HT (4)	.88	3.02	3.2	2.22	.43
8-way Opteron (8)	1.0	5.29	5.73	4.53	2.03
8-core Niagara (32)	.98	10.46	17.21	15.34	6.49

Under the hood

- Already discussed naïve implementation – use Thread
  - Problem is it uses a lot of threads, and they mostly just wait around
- Executor is similarly a bad choice
  - Likely deadlock if pool is bounded – standard thread pools are designed for *independent* tasks
  - Standard thread pools can have high contention for task queue and other data structures when used with fine-grained tasks
- An ideal solution minimizes
  - Context switch overhead between worker threads
    - Have as many threads as hardware threads, and keep them busy
  - Contention for data structures
    - Avoid a common task queue

Work stealing

- Fork-join framework is implemented using *work-stealing*
  - Create a limited number of worker threads
  - Each worker thread maintains a private double-ended work queue (deque)
    - Optimized implementation, not the standard JUC deques
  - When forking, worker pushes new task at the *head* of its deque
  - When waiting or idle, worker pops a task off the *head* of its deque and executes it
    - Instead of sleeping
  - If worker's deque is empty, steals an element off the *tail* of the deque of another randomly chosen worker

Work stealing

- Work-stealing is efficient – introduces little per-task overhead
- Reduced contention compared to shared work queue
  - No contention ever for head
    - Because only the owner accesses the head
  - No contention ever between head and tail access
    - Because good queue algorithms enable this
  - Almost never contention for tail
    - Because stealing is infrequent, and steal collisions more so
- Stealing is infrequent
  - Workers put and retrieve items from their queue in LIFO order
  - Size of work items gets smaller as problem is divided
  - So when a thread steals from the tail of another worker's queue, it generally steals a big chunk!
    - This will keep it from having to steal again for a while

Work stealing

- When pool.invoke() is called, task is placed on a random deque
  - That worker executes the task
    - Usually just pushes two more tasks onto its deque – very fast
    - Starts on one of the subtasks
  - Soon some other worker steals the other top-level subtask
  - Pretty soon, most of the forking is done, and the tasks are distributed among the various work queues
  - Now the workers start on the meaty (sequential) subtasks
    - If work is unequally distributed, corrected via stealing
- Result: reasonable load balancing
  - With no central coordination
  - With little scheduling overhead
  - With minimal synchronization costs
    - Because synchronization is almost never contended

Example: Traversing and marking a graph

- Extend LinkedAsyncAction instead of RecursiveAction
  - LinkedAsyncAction manages parent-child relationship
  - Finish method means “wait for all my children”
    - Example uses AtomicBoolean to safely manage shared mark bits

```
class GraphVisitor extends LinkedAsyncAction {
    GraphVisitor(GraphVisitor parent, Node node) {
        super(parent); this.node = node;
    }
    protected void compute() {
      if (node.mark.compareAndSet(false, true)) {
         for (Edge e : node.edges()) {
            Node dest = e.getDestination();
            if (!dest.mark.get())
               new GraphVisitor(this, dest).fork();
         }
         visit(node);
      }
      finish();
   }
}
```

Other applications

- Fork-join can be used for parallelizing many types of problems
  - Matrix operations
    - Multiplication, LU decomposition, etc
  - Finite-element modeling
  - Numerical integration
  - Game playing
    - Move generation
    - Move evaluation
    - Alpha-beta pruning

Taking it up a level

- Still lots of “boilerplate” code in fork-join tasks
  - Decomposing into subproblems, choosing between recursive and sequential execution, managing subtasks
- Would be nicer to specify parallel aggregate operations at a higher abstraction level
  - Enter *ParallelArray*
- The ParallelArray classes let you declaratively specify aggregate operations on data arrays
  - And uses fork-join to efficiently execute on the available hardware
- Versions for primitives and objects
  - ParallelArray<T>, ParallelLongArray, etc
- Resembles a restricted, in-memory, parallel DBMS
  - Less powerful than LinQ, but designed for parallelization with a transparent cost model

ParallelArray

- Coding select-max with ParallelArray is trivial

```
ParallelLongArray pa
    = ParallelLongArray.createUsingHandoff(array, fjPool);
long max = pa.max();
```

- ParallelArray framework automates fork-join decomposition for operations on arrays
  - Supports filtering, element mapping, and combination across multiple parallel arrays
  - Batches all operations into a single parallel step

ParallelArray

- Slightly less trivial example: select highest GPA of students graduating this year

```
class Student {
    String name;
    int graduationYear;
    double gpa;
}
ParallelArray<Student> students
    = ParallelArray.createUsingHandoff(studentsArray, forkJoinPool);
double highestGpa = students.withFilter(graduatesThisYear)
                            .withMapping(selectGpa)
                            .max();
Ops.Predicate<Student> graduatesThisYear = new Ops.Predicate<Student>() {
    public boolean op(Student s) {
        return s.graduationYear == 2008;
    }
};
Ops.ObjectToDouble<Student> selectGpa = new Ops.ObjectToDouble<Student>()
{
    public double op(Student student) {
        return student.gpa;
    }
};
```

ParallelArray

- We specify three operations – filter, map, aggregate
  - Uses filtering to select students graduating this year
  - Uses mapping to select each student's GPA
  - Applies max() to result
- Query *looks* imperative, but in fact is more like declarative
  - The actual work isn't done until the aggregation step (max())
    - Other methods merely set up the "query"
    - Filtering and mapping calls just set up lightweight descriptors

ParallelArray

- There are some restrictions
  - Filtering must precede mapping
  - Mapping must precede aggregation
  - Must have an aggregation or replacement step
    - Because that’s where the work is done
    - Can use all() method to return a ParallelArray containing all filtered rows
- These restrictions are largely in aid of maintaining a transparent cost model
  - SQL let’s you express arbitrarily complicated queries in a single statement, but it is harder to predict their performance
  - ParallelArray makes it much more obvious how much the query is going to cost

ParallelArray example: mean and variance

- We can use ParallelArray to sample, compute mean and variance in three parallel operations
  - Separate operations needed because of dataflow dependencies

```
ParallelDoubleArray data
   = ParallelDoubleArray.create(SIZE, forkJoinPool)
       .replaceWithGeneratedValue(getSample);
final double mean = data.sum() / SIZE;
double variance = data
       .withMapping(new Ops.DoubleOp() {
                       public double op(double v) {
                          return (v-mean) * (v-mean);
                       }
                    })
       .sum() / SIZE;
```

ParallelArray

- Basic operations supported by ParallelArray
  - Filtering – select a subset of the elements
    - Can specify multiple filters
    - Binary search supported on sorted parallel arrays
  - Mapping – convert selected elements to another form
    - Such as selecting a student's GPA
  - Replacement – create a new ParallelArray derived from the original
    - Sorting, running accumulation
  - Aggregation – combine all values into a single value
    - Maxima, minima, sum, average
    - General-purpose reduce() method
  - Application – perform an action for each selected element

Combining multiple ParallelArrays

- Operations can combine multiple parallel ParallelArrays
  - Compute min(a[i] + b[i] + c[i])

```
// a, b, and c are ParallelLongArrays
 long minSum = a.withMapping(CommonOps.longAdder(), b)
                .withMapping(CommonOps.longAdder(), c)
                .min();
```

  - CommonOps has combiners for arithmetic operations, max and min, etc
  - This form of withMapping uses the combiner to combine the element of the receiver array with the corresponding element of the other array

Connection with closures

- One of the features proposed for JDK 7 is *closures*
  - One goal of closures is to reduce redundant boilerplate code
  - Ugliest part of ParallelArray is the helper methods like selectGpa()
  - These would go away with closures

```
double highestGpa = students
    .withFilter( { Student s => (s.graduationYear == THIS_YEAR) } )
    .withMapping( { Student s => s.gpa } )
    .max();
```

- With closures, API could be written in terms of function types instead of named types
  - **`Ops.Predicate<T>`** becomes **`{ T => boolean }`**
  - Which might be a benefit or a disadvantage
    - Names are useful

From Concurrent to Parallel
Library-based parallelism in JDK 7

Brian Goetz

Sr. Staff Engineer, Sun Microsystems
brian.goetz@sun.com