

ORACLE[®]



Data Parallelism in Java

Brian Goetz
Java Language Architect

Hardware trends

(Graphic courtesy Herb Sutter)

As of ~2003, we stopped seeing increases in CPU clock rate

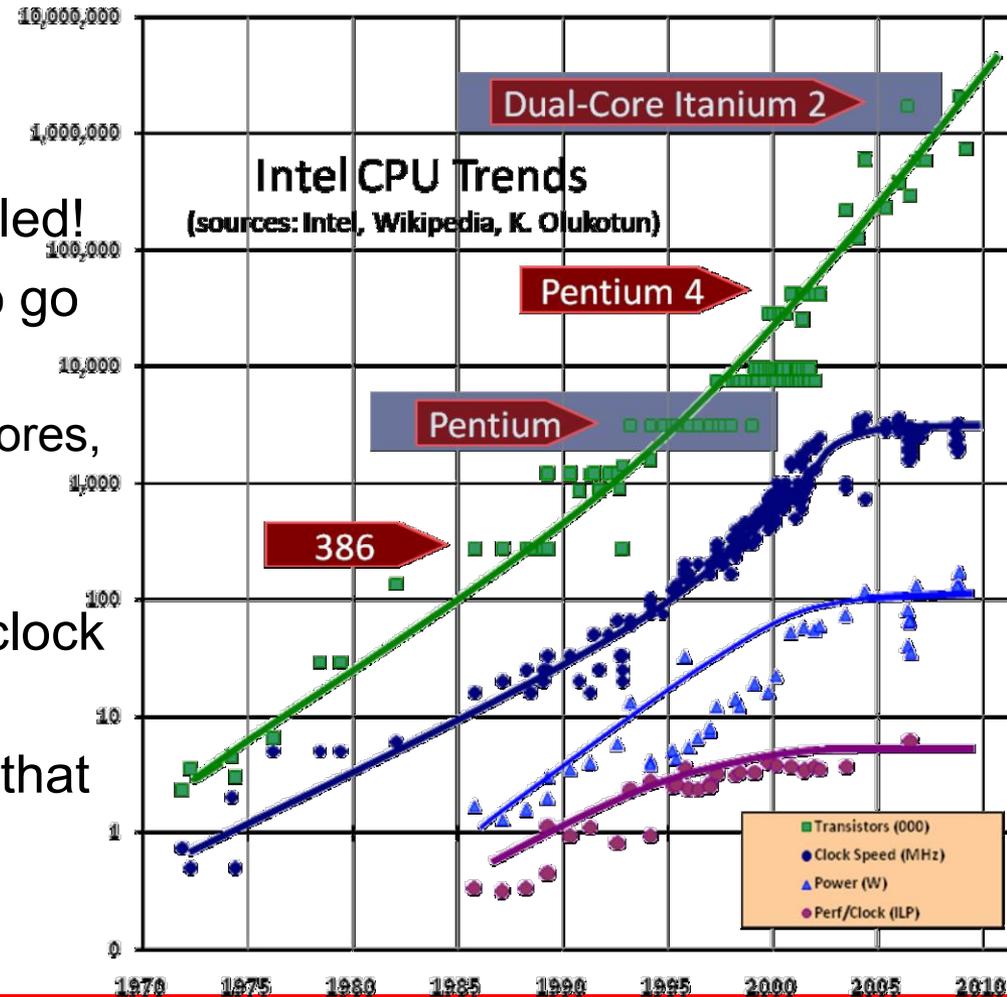
Moore's law has not been repealed!

Chip designers have nowhere to go but parallel

Moore's Law now gives more cores, not faster cores

Hit the wall in power dissipation, instruction-level parallelism, clock rate, and chip scale

We must learn to write software that parallelizes gracefully



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

The future – 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

We need to learn to spot latent parallelism

No single technique works in all situations

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

One 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

Point solutions

Work queues + thread pools
Divide and conquer (fork-join)
Parallel collection libraries
Map/Reduce
Actors / Message passing
Software Transactional Memory (STM)
GPU-based computation

Point solution: Thread pools / work queues

A reasonable solution for coarse-grained concurrency

Typical server applications with medium-weight requests

Database servers

File servers

Web servers

Library support added in JDK 5

Works well in SMP systems

Even when tasks do IO

Shared work queue is eventually source of contention

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 has 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

Point solution: divide and conquer

Divide-and-conquer breaks down a problem into subproblems, solves the subproblems, and combines the result

- Apply recursively until subproblems are so small that sequential solution is faster

- Scales well – can keep 100s of CPUs busy

- Good for fine-grained tasks

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>

Fork-join libraries: coming in JDK 7

There are good libraries for fork-join decomposition

One such library is Doug Lea's "jsr166y" library

- Scheduled for inclusion in JDK 7

- Also can be used with JDK 5, 6 as a standalone library

Solving select-max with fork-join

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

forkJoin() creates two new tasks and waits for them

ForkJoinPool is like an Executor, but optimized for fork-join task

Waiting for other pool tasks risks *thread-starvation*
deadlock in standard executors

While waiting for the results of a task, pool threads find other tasks to work on

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

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

```
class GraphVisitor extends RecursiveAction {
    private final Node node;
    private final Collection<ForkJoinTask> children =
        new ArrayList<>();

    GraphVisitor(Node node) {
        this.node = node;
    }

    protected void compute() {
        if (node.mark.compareAndSet(false, true)) {
            // Do node-visiting action here
            for (Edge e : node.edges()) {
                Node dest = e.getDestination();
                if (!dest.mark.get()) {
                    children.add(new GraphVisitor(dest));
                }
            }
            ForkJoinTask.invokeAll(children);
        }
    }
}
```

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

Point solution: parallel collection libraries

One can build on the fork/join approach to add parallel aggregate operations to collection-like classes

Doug Lea's `ParallelArray` (from `extra166y`) is one example

Collections will likely acquire bulk data operations in JDK 8

Aim is to enable code that has a functional / query-like feel

Example: ParallelArray

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

ParallelArray<Student> students
    = ParallelArray.createUsingHandoff(studentsArray, forkJoinPool);

double highestGpa = students.withFilter(new Ops.Predicate<Student>() {
    public boolean op(Student s) {
        return s.graduationYear == 2010;
    }
})
    .withMapping(new Ops.ObjectToDouble<Student>() {
    public double op(Student student) {
        return student.gpa;
    }
})
    .max();
```

ParallelArray

Bulk data operations offer opportunities for library-directed parallelism and laziness

More functional style, code reads more like problem statement

Except that inner classes make it *painful*

With closures in the language (JDK 8), gets much better:

```
double highestGpa
  = students.withFilter(#{ s -> s.graduationYear == 2010 })
             .withMapping(#{ s -> student.gpa })
             .max();
```

Point solution: Map / Reduce

Map / Reduce is a distributed generalization of fork/join

Decomposes data queries across a cluster

- Designed for very large input data sets (usually distributed)

- “Map” tasks each search a portion of the data set

 - Map task often runs on node where the data is, for locality

- “Reduce” tasks combine results of map tasks

- Framework handles distribution, reliability, scheduling

Scales to thousands of nodes

High quality open-source implementations available

(e.g., Hadoop)

Point solution: Actors

Actors are a computing model where state is not shared – all mutable state is confined to actors

Actors communicate by sending messages to each other

To access another actor's state, send it a request, and it sends a response, containing a *read-only copy* of the state

No shared mutable state → less error-prone

Immutable messages → easy distribution across cluster

Works well in Erlang and Scala

Possible in Java, but clunkier and requires more discipline

Point solution: Software Transactional Memory

STM has been sold as “garbage collection for concurrency”

Some might say oversold

Programmer demarcates transaction boundaries

System figures out what is modified, and prevents interference

Performance of current general-purpose systems is poor

But...seems to work very well in Clojure

Because Clojure is mostly functional and greatly limits mutable state

Point solution: GPU computing

GPUs have zillions of simple cores

Great for doing the same operation to lots of data (SIMD)

Designed for graphics, but can be programmed for general-purpose computations too

“Vector processors on steroids”

Significant latency moving data between CPU / GPU

But enough parallelism makes up for that

Works well on certain types of problems

Widely supported APIs – CUDA, OpenCL,
DirectCompute

Java bindings starting to appear – jCuda, JOCL

Conclusion

We have not yet identified a general framework for parallel computing in Java

But there are lots of point solutions that work on specific problem types

Each point solution introduces a specific style of programming and program organization