IBM

developerWorks®

Java theory and practice: Stick a fork in it, Part 1

Learn how to exploit fine-grained parallelism using the fork-join framework coming in Java 7

Brian Goetz

November 13, 2007

One of the additions to the `java.util.concurrent` packages coming in Java™ 7 is a framework for fork-join style parallel decomposition. The fork-join abstraction provides a natural mechanism for decomposing many algorithms to effectively exploit hardware parallelism. The next installment in this series covers the `ParallelArray` classes, which simplify parallel sorting and searching operations on in-memory data structures.

View more content in this series

Hardware trends drive programming idioms

Languages, libraries, and frameworks shape the way we write programs. Even though Alonzo Church showed in 1934 that all the known computational frameworks were equivalent in the set of programs they could represent, the set of programs that real programmers actually write is shaped by the idioms that the programming model — driven by languages, libraries, and frameworks — makes easy to express.

In turn, the dominant hardware platforms of the day shape the way we create languages, libraries, and frameworks. The Java language has had support for threads and concurrency from day 1; the language includes synchronization primitives such as `synchronized` and `volatile`, and the class library includes classes such as `Thread`. However, the concurrency primitives that were sensible in 1995 reflected the hardware reality of the time: most commercially available systems provided no parallelism at all, and even the most expensive systems provided only limited parallelism. In those days, threads were used primarily for expressing *asynchrony*, not *concurrency*, and as a result, these mechanisms were generally adequate to the demands of the time.

As multiprocessor systems started to become cheaper, more applications needed to exploit the hardware parallelism they provided, and programmers found that writing concurrent programs using the low-level primitives provided by the language and class library was difficult and error-prone. In Java 5, the `java.util.concurrent` package was added to the Java platform, providing a set of useful components for building concurrent applications: concurrent collections, queues, semaphores, latches, thread pools, and so on. These mechanisms were well-suited to programs with a reasonably coarse task granularity; applications needed only to partition their work so that the number of concurrent tasks was not consistently less than the (small) number of processors available. Using the processing of a single request as the unit of work in a Web server, mail server, or database server, applications generally met that requirement, and so these mechanisms were sufficient to keep modestly parallel hardware sufficiently utilized.

Going forward, the hardware trend is clear; Moore's Law will not be delivering higher clock rates, but instead delivering more cores per chip. It is easy to imagine how you can keep a dozen processors busy using a coarse-grained task boundary such as a user request, but this technique will not scale to thousands of processors — traffic may scale exponentially for short periods of time, but eventually the hardware trend wins out. As we enter the many-core era, we will need to find finer-grained parallelism or risk keeping processors idle even though there is plenty of work to do. As the dominant hardware platform shifts, so too must the software platform if we wish to keep up. To this end, Java 7 will include a framework for representing a certain class of finer-grained parallel algorithms: the *fork-join framework*.

Exposing finer-grained parallelism

Most server applications today use the user request-response processing as their unit of work. Server applications typically run many more concurrent threads, or requests, than there are processors available. The reason is because in most server applications, the processing of a request includes a fair amount of I/O, which does not require very much of the processor. (All network server applications do a lot of socket I/O, as requests are received via sockets; many do a fair amount of disk (or database) I/O as well.) If each task spends 90 percent of its time waiting for I/O to complete, you'll need 10 times as many concurrent tasks as processors to keep the processors fully utilized. As processor counts increase, there may not be enough concurrent requests to keep all the processors busy. However, it is still possible to use parallelism to improve another measure of performance: the amount of time the user has to wait to get a response.

As an example of a typical network server application, consider a database server. When a request arrives at the database server, it goes through a series of phases. First the SQL statement is parsed and validated. Then a query plan must be selected; for complicated queries, database servers may evaluate many different candidate plans to minimize the expected number of I/O operations. Searching for a query plan can be a CPU-intensive task; at some point, considering more candidate plans will reach a point of negative returns, but evaluating too few candidate plans will almost certainly require more I/O than necessary. After the data is retrieved from disk, more processing may be required on the resulting data set; the query may include aggregate operations such as SUM or AVERAGE or may require sorting of the data set. Then the result must be encoded and returned to the requestor.

Just like most server requests, processing an SQL query involves a mixture of computation and I/O. No amount of additional CPU power can reduce the time it takes for an I/O to complete (though you can use additional *memory* to reduce the number of I/Os by caching the results of previous I/O operations), but you can shorten the amount of time it takes for the CPU-intensive portions of the request processing (such as plan evaluation and sorting) by parallelizing them. In evaluating candidate query plans, different plans can be evaluated concurrently; in sorting data sets, a large data set can be broken down into smaller data sets, individually sorted in parallel, and then merged. Doing so improves the user's perception of performance (even though it may require more total work to be performed to service the request) because results are received faster.

Divide and conquer

Merge sort is an example of a *divide-and-conquer* algorithm, where a problem is recursively broken down into subproblems, and the solutions to the subproblems are combined to arrive at the final result. Divide-and-conquer algorithms are often useful in sequential environments but can become even more effective in parallel environments because the subproblems can often be solved concurrently.

A typical parallel divide-and-conquer algorithm takes the form shown in Listing 1:

Listing 1. Pseudo-code for generic divide-and-conquer parallel algorithms

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

The first thing a parallel divide-and-conquer algorithm does is evaluate whether the problem is so small that a sequential solution would be faster; typically, this is done by comparing the problem size to some threshold. If the problem is large enough to merit parallel decomposition, it divides the problem into two or more sub-problems and recursively invokes itself on the sub-problems in parallel, waits for the results of the sub-problems, and then combines the results. The ideal threshold for choosing between sequential and parallel execution is a function of the cost of coordinating the parallel tasks. If coordination costs are zero, a larger number of finer-grained tasks tend to offer better parallelism; the lower the coordination costs, the finer-grained we can go before we need to switch to a sequential approach.

Fork-join

The example in Listing 1 makes use of a nonexistent INVOKE-IN-PARALLEL operation; its behavior is that the current task is suspended, the two subtasks are executed in parallel, and the current task waits until they complete. Then the results of the two subtasks can be combined. This kind of parallel decomposition is often called *fork-join* because executing a task *forks* (starts) multiple subtasks and then *joins* (waits for completion) with them.

Listing 2 shows an example of a problem that is suitable to a fork-join solution: searching a large array for its maximal element. Of course, this is a very simple example, but the fork-join technique is suitable for a wide variety of searching, sorting, and data analysis problems.

Listing 2. Selecting a maximal element from a large array

```
public class SelectMaxProblem {
    private final int[] numbers;
    private final int start;
    private final int end;
    public final int size;

    // constructors elided

    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i=start; i<end; i++) {
            int n = numbers[i];
            if (n > max)
                max = n;
        }
        return max;
    }

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

Note that the `subproblem()` method does not copy the elements; it merely copies the array reference and offsets into an existing data structure. This is typical of fork-join problem implementations because the process of recursively dividing the problem will create a potentially large number of new `Problem` objects. In this case, the data structure being searched is not modified at all by the searching tasks, so there is no need to maintain private copies of the underlying data set for each task, and therefore no need to incur the extra overhead of copying.

Listing 3 illustrates a solution for `SelectMaxProblem` using the fork-join package that is scheduled for inclusion in Java 7. The package is being developed openly by the JSR 166 Expert Group, using the code name jsr166y, and you can download it separately and use it with Java 6 or later. (It will eventually live in the package `java.util.concurrent.forkjoin`.) The operation `invoke-in-parallel` is implemented by the `coInvoke()` method, which invokes multiple actions simultaneously and waits for them all to complete. A `ForkJoinExecutor` is like an `Executor` in that it is designed for running tasks, except that it specifically designed for computationally intensive tasks that do not ever block except to wait for another task being processed by the same `ForkJoinExecutor`.

The fork-join framework supports several styles of `ForkJoinTasks`, including those that require explicit completions and those executing cyclically. The `RecursiveAction` class used here directly supports the style of parallel recursive decomposition for non-result-bearing tasks; the `RecursiveTask` class addresses the same problem for result-bearing tasks. (Other fork-join task classes include `CyclicAction`, `AsyncAction`, and `LinkedAsyncAction`; see the Javadoc for more details on how they are used.)

Listing 3. Solving the select-max problem with the fork-join framework

```
public class MaxWithFJ extends RecursiveAction {
    private final int threshold;
    private final SelectMaxProblem problem;
    public int result;

    public MaxWithFJ(SelectMaxProblem problem, int threshold) {
        this.problem = problem;
        this.threshold = threshold;
    }

    protected void compute() {
        if (problem.size < threshold)
            result = problem.solveSequentially();
        else {
            int midpoint = problem.size / 2;
            MaxWithFJ left = new MaxWithFJ(problem.subproblem(0, midpoint), threshold);
            MaxWithFJ right = new MaxWithFJ(problem.subproblem(midpoint +
              1, problem.size), threshold);
            coInvoke(left, right);
            result = Math.max(left.result, right.result);
        }
    }

    public static void main(String[] args) {
        SelectMaxProblem problem = ...
        int threshold = ...
        int nThreads = ...
        MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
        ForkJoinExecutor fjPool = new ForkJoinPool(nThreads);

        fjPool.invoke(mfj);
        int result = mfj.result;
    }
}
```

Table 1 shows some results of selecting the maximal element of a 500,000 element array on various systems and varying the threshold at which the sequential version is preferred to the parallel version. For most runs, the number of threads in the fork-join pool was equal to the number of hardware threads (cores times threads-per-core) available. The numbers are presented as a speedup relative to the sequential version on that system.

Table 1. Results of Running select-max on 500k-element Arrays on various systems

	Threshold=500k	Threshold=50k	Threshold=5k	Threshold=500	Threshold=-50
Pentium-4 HT (2 threads)	1.0	1.07	1.02	.82	.2
Dual-Xeon HT (4 threads)	.88	3.02	3.2	2.22	.43
8-way Opteron (8 threads)	1.0	5.29	5.73	4.53	2.03
8-core Niagara (32 threads)	.98	10.46	17.21	15.34	6.49

The results are quite encouraging in that they show good speedups over a broad range of parameters. So as long as you avoid choosing completely unreasonable parameters for the problem or the underlying hardware, you will get good results with little tuning. With chip-multithreading, it is not clear what an optimal speedup should be; clearly CMT approaches like Hyperthreading deliver less performance than the equivalent number of actual cores, but just how much less is going to depend on a wide range of factors, including the cache miss rate of the code being executed.

The sequential thresholds chosen here range from 500K (the size of the array, meaning effectively no parallelism) down to 50. A sequential threshold of 50 in this case is well into "ridiculously small" territory, and the results show that with a ridiculously low sequential threshold, the overhead of managing the fork-join tasks dominates. But they do show that as long as you avoid "ridiculously high" and "ridiculously low" parameters, you can get good results without tuning. Choosing `Runtime.availableProcessors()` as the number of worker threads generally offers close to optimal results, as tasks executed in fork-join pools are supposed to be CPU-bound, but again, results tend to not be very sensitive to this parameter so long as you avoid sizing the pool way too large or way too small.

No explicit synchronization is required in the `MaxWithFJ` class. The data it operates on is constant for the lifetime of the problem, and there is sufficient internal synchronization within the `ForkJoinExecutor` to guarantee visibility of the problem data to subtasks, as well as to guarantee visibility of subtask results to the tasks that join with them.

Anatomy of the fork-join framework

A fork-join framework like the one illustrated in Listing 3 can be implemented in many ways. Using raw threads is a possibility; `Thread.start()` and `Thread.join()` provide all the necessary functionality. However, this approach may require more threads than the VM can support. For a problem size of *N* (assuming a very small sequential threshold), *O(N)* threads would be required to solve the problem (the problem tree has depth log_2N, and a binary tree of depth *k* has 2^k nodes). And, of those, half would spend almost their entire lives waiting for subtasks to complete. Threads are expensive to create and use a lot of memory, making this approach prohibitive. (While this approach can be made to work, the code is more complicated and it requires very careful tuning of the parameters for the problem size and hardware.)

Using conventional thread pools to implement fork-join is also challenging because fork-join tasks spend much of their lives waiting for other tasks. This behavior is a recipe for *thread starvation deadlock*, unless the parameters are carefully chosen to bound the number of tasks created or the pool itself is unbounded. Conventional thread pools are designed for tasks that are independent of each other and are also designed with potentially blocking, coarse-grained tasks in mind — fork-join solutions produce neither. Fine-grained tasks in conventional thread pools can also generate excessive contention for the task queue shared among all workers.

Work stealing

The fork-join framework reduces contention for the work queue by using a technique known as *work stealing*. Each worker thread has its own work queue, which is implemented using a double-ended queue, or *deque*. (Java 6 adds several deque implementations to the class library, including `ArrayDeque` and `LinkedBlockingDeque`.) When a task forks a new thread, it pushes it onto the head of its own deque. When a task executes a join operation with another task that has not yet completed, rather than sleeping until the target task is complete (as `Thread.join()` would), it pops another task off the head of its deque and executes that. In the event the thread's task queue is empty, it then tries to steal another task off the *tail* of another thread's deque.

Work stealing can be implemented with standard queues, but using a deque has two principle advantages over a standard queue: reduced contention and reduced stealing. Because only the worker thread ever accesses the head of its own deque, there is never contention for the head of a deque; because the tail of the deque is only ever accessed when a thread runs out of work, there is rarely contention for the tail of any thread's deque either. (The deque implementation incorporated into the fork-join framework exploits these access patterns to further reduce the costs of coordination.) This reduction in contention dramatically reduces the synchronization costs compared to a traditional thread-pool-based approach. Furthermore, the last-in-first-out (LIFO) ordering of tasks implied by such an approach means that the largest tasks sit at the tail of the deque, and therefore when another thread has to steal a task, it steals a large one that can be decomposed into smaller ones, reducing the need to steal again in the near future. Work stealing thus produces reasonable load balancing with no central coordination and minimal synchronization costs.

Summary

The fork-join approach offers a portable means of expressing a parallelizable algorithm without knowing in advance how much parallelism the target system will offer. All sorts of sorting, searching, and numerical algorithms are amenable to parallel decomposition. (In the future, standard library mechanisms like `Arrays.sort()` may become consumers of the fork-join framework, allowing applications to get some of the benefits of parallel decomposition for free.) As processor counts continue to increase, we will need to expose more parallelism inherent in our programs to utilize these processors effectively; parallel decomposition of computationally intensive activities such as sorting makes it easier for programs to take advantage of tomorrow's hardware.

Related topics

- "Java theory and practice: Stick a fork in it, Part 2" (developerWorks, March 2008): Explore the `ParallelArray` classes, which simplify parallel sorting and searching operations on in-memory data structures.
- "Church-Turing thesis": States that all known nontrivial computational models are equally expressive.
- Merge sort: Another example of a divide and conquer algorithm.
- Section 4.4 of *Concurrent Programming in Java* (Doug Lea, Prentice Hall PTR, November 1999): Covers parallel decomposition in greater detail.
- Doug Lea's concurrency-interest website: Download the fork-join framework as part of the jsr166y package, or read the paper on its design.
- Section 8.1 of *Java Concurrency in Practice* (Brian Goetz, Addison-Wesley Professional, May 2006): Covers thread-starvation deadlock.