Java theory and practice: Stick a fork in it, Part 2

Accelerate sorting and searching with the ParallelArray classes in Java 7

Brian Goetz

March 04, 2008

One of the additions to the `java.util.concurrent` packages coming in Java™ 7 is a library for fork-join-style parallel decomposition. In part one of this series, author Brian Goetz showed how fork-join provides a natural mechanism for decomposing many algorithms to effectively exploit hardware parallelism. In this article, he'll cover the `ParallelArray` classes, which simplify parallel sorting and searching operations on in-memory data structures.

View more content in this series

In the last installment of *Java theory and practice*, we examined the fork-join library, which will be added to the `java.util.concurrent` package in Java 7. Fork-join is a technique that makes it easy to express divide-and-conquer parallel algorithms in a way that admits efficient execution on a wide range of hardware, without code changes.

To effectively utilize the available hardware as processor counts increase, we are going to have to identify and exploit finer-grained parallelism in our programs. In recent years, choosing coarse-grained task boundaries (such as processing a single request in a Web application) and executing tasks in thread pools often provided sufficient parallelism to achieve acceptable hardware utilization. But moving forward, we're going to have to dig deeper to find enough parallelism to keep the hardware busy. One area that is ripe for parallelization is sorting and searching in large data sets. It's easy to express such problems using fork-join, as you saw in the previous installment. But because these problems are so common, the class library provides an even easier way —`ParallelArray`.

Review: fork-join decomposition

Fork-join embodies the technique of divide-and-conquer; take a problem and recursively break it down into subproblems until the subproblems are small enough that they can be more effectively solved sequentially. The recursive step involves dividing a problem into two or more subproblems, queueing the subproblems for solution (the *fork* step), waiting for the results of the subproblems (the *join* step), and merging the results. One example of such an algorithm is merge-sort, illustrated in Listing 1, using the fork-join library:

Listing 1. Merge-sort using the fork-join library

```
public class MergeSort extends RecursiveAction {
    final int[] numbers;
    final int startPos, endPos;
    final int[] result;

    private void merge(MergeSort left, MergeSort right) {
        int i=0, leftPos=0, rightPos=0, leftSize = left.size(), rightSize = right.size();
        while (leftPos < leftSize && rightPos < rightSize)
            result[i++] = (left.result[leftPos] <= right.result[rightPos])
                    ? left.result[leftPos++]
                    : right.result[rightPos++];
        while (leftPos < leftSize)
            result[i++] = left.result[leftPos++];
        while (rightPos < rightSize)
            result[i++] = right.result[rightPos++];
    }

    public int size() {
        return endPos-startPos;
    }

    protected void compute() {
        if (size() < SEQUENTIAL_THRESHOLD) {
            System.arraycopy(numbers, startPos, result, 0, size());
            Arrays.sort(result, 0, size());
        }
        else {
            int midpoint = size() / 2;
            MergeSort left = new MergeSort(numbers, startPos, startPos+midpoint);
            MergeSort right = new MergeSort(numbers, startPos+midpoint, endPos);
            coInvoke(left, right);
            merge(left, right);
        }
    }
}
```

Merge-sort is not an inherently parallel algorithm, as it can be done sequentially, and is popular when the data set is too large to fit in memory and must be sorted in pieces. Merge sort has *O(n log n)* worst-case and average-case performance. But because the merging is difficult to do in place, generally, it has higher memory requirements than in-place sort algorithms, such as quick-sort. But because the sorting of the subproblems can be done in parallel, it parallelizes better than quick-sort.

Given a fixed number of processors, parallelization still can't turn an *O(n log n)* problem into an *O(n)* one, but the more amenable the problem to parallelization, the closer to a factor of n_{cpus} by which parallelization can reduce the total runtime. Reducing the total runtime means that the user gets the result sooner — even if it takes more total CPU cycles to do the work in parallel than it does sequentially.

The principal benefit of using the fork-join technique is that it affords a portable means of coding algorithms for parallel execution. The programmer does not have to be aware of how many CPUs will be available in deployment; the runtime can do a good job of balancing work across available workers, yielding reasonable results across a wide range of hardware.

Fine-grained parallelism

The most accessible (and alliterative) source of finer-grained parallelism in mainstream server applications is sorting, searching, selection, and summarizing of data sets. Each of these problems can be easily parallelized using divide-and-conquer, and can be easily represented as fork-join tasks. For example, to parallelize the averaging of a large data set, you can recursively break it down into smaller data sets — as was done in merge-sort — average the subsets, and the combination step simply computes a weighted average of the averages of the subsets.

`ParallelArray`

For sorting and searching problems, the fork-join library gives you an even easier means of expressing parallelizable operations on data sets: the `ParallelArray` classes. The idea is that a `ParallelArray` represents a collection of structurally similar data items, and you use the methods on `ParallelArray` to create a description of how you want to slice and dice the data. You then use the description to actually execute the array operations (which uses the fork-join framework under the hood) in parallel. This approach has the effect of letting you declaratively specify data selection, transformation, and post-processing operations, and letting the framework figure out a reasonable parallel execution plan, just as database systems allow you to specify data operations in SQL and hide the mechanics of how the operations are implemented. Several implementations of `ParallelArray` are available for different data types and sizes, including for arrays of objects and for arrays of various primitives.

Listing 2 shows an example of using `ParallelArray` to summarize student grades, illustrating the basic operations of selection, projection, and summarization. The `Student` class contains information about students (name, graduation year, GPA). The helper object `isSenior` is used to select only the students graduating this year, and the helper object `getGpa` extracts the GPA field from a given student. The expression at the beginning of the listing creates a `ParallelArray` representing a set of students and then uses it to select the best GPA from the students graduating this year.

Listing 2. Using `ParallelArray` to select, process, and summarize data

```
ParallelArray<Student> students = new ParallelArray<Student>(fjPool, data);
double bestGpa = students.withFilter(isSenior)
                         .withMapping(selectGpa)
                         .max();

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

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

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

The code for expressing operations on parallel arrays is somewhat deceptive. The `withFilter()` and `withMapping()` methods do not actually search or transform the data; they merely set up the parameters of the "query." The actual work is done in the final step, in this case, the call to `max()`.

The basic operations supported by `ParallelArray` are as follows:

- **Filtering:** Selecting a subset of the elements to be involved in a computation. In Listing 2, the filter was specified with the `withFilter()` method.
- **Application:** Applying a procedure to each selected element. Listing 2 doesn't show this technique, but the `apply()` method allows you to perform an action for each selected element.
- **Mapping:** Converting selected elements to another form (such as extracting a data field from the element). This conversion is done in the example by the `withMapping()` method, where we convert `Student` to the student's GPA. The result is a `ParallelArray` of the results of the specified selection and mapping.
- **Replacement:** Creating a new parallel array by replacing each element with another derived from it. This technique is similar to mapping, but produces a new `ParallelArray` on which further queries may be performed. One case of replacement is sorting, where elements are replaced with different elements so that the resulting array is in sorted order (the built-in `sort()` method is provided for this action). Another special case is the `cumulate()` method, which replaces each element with a running accumulation according to a specified combination operation. Replacement can also be used to combine multiple `ParallelArray`s, such as creating a `ParallelArray` whose elements are the values of `a[i]+b[i]` for parallel arrays `a` and `b`.
- **Summarization:** Combining all the values into a single value, such as computing a sum, average, minimum, or maximum. The example in Listing 2 uses the `max()` method. The predefined summarization methods, such as `min()`, `sum()`, and `max()`, are built using the more general-purpose `reduce()` method.

Common summary operations

In Listing 2, we were able to specify the calculation of the highest GPA of any student, but what we probably want is something slightly different — which student had the highest GPA. This calculation could be done using two computations (one to compute the highest GPA, and another to select the students who have that GPA), but `ParallelArray` provides an easier means of getting at commonly desired summary statistics such as maximum, minimum, sum, average, and the index of the maximal and minimal elements. The `summary()` method computes these summary statistics in a single, parallel operation.

Listing 3 shows the `summary()` method computing the summary statistics, which include the indexes of the minimal and maximal elements, to avoid having to make multiple passes on the data:

Listing 3. Finding the student with the highest GPA

```
SummaryStatistics summary = students.withFilter(isSenior)
                            .withMapping(selectGpa)
                            .summary();
System.out.println("Worst student: " + students.get(summary.minIndex()).name;
System.out.println("Average GPA: " + summary.getAverage();
```

Limitations

`ParallelArray` is not intended to be a general-purpose in-memory database, nor a general-purpose mechanism for specifying data transformations and extractions (like the language-integrated query — LinQ — features in .NET 3.0); it is intended only to simplify the expression of a specific range of data selection and transformation operations so that they can be easily and automatically parallelized. Accordingly, it has several limitations; for example, filter operations must be specified before mapping operations. (Multiple filter operations are permitted, though it is often more efficient to combine them into a single compound-filter operation.) Its main purpose is to free developers from thinking about how the work can be parallelized; if you can express the transformation in terms of the operations provided by `ParallelArray`, you should get reasonable parallelization for no effort.

Performance

To assess the effectiveness of `ParallelArray`, I wrote a simple, unscientific program that runs the query for various sizes of array and fork-join pools. The results were run on a Core 2 Quad system running Windows. Table 1 shows the speedup relative to the base case (1000 students, one thread):

Table 1. Performance measurement for the max-GPA query

Students	Threads			
	1	2	4	8
1000	1.00	0.30	0.35	1.20
10000	2.11	2.31	1.02	1.62
100000	9.99	5.28	3.63	5.53
1000000	39.34	24.67	20.94	35.11
10000000	340.25	180.28	160.21	190.41

While the results are fairly noisy (biased by a number of factors, including GC activity), you can see that not only are the best results achieved with a pool size equal to the number of cores available (which you would expect, given that the tasks are entirely compute-bound), but that also we've achieved a speedup of 2-3x with four cores compared to one, showing that it is possible to get reasonable parallelism using high-level, portable mechanisms without tuning.

Connection with closures

`ParallelArray` offers a nice way to declaratively specify filtering, processing, and aggregation operations on data sets, while also facilitating automatic parallelization. However, even though the syntax is easier to express than using the raw fork-join library, the syntax is still somewhat cumbersome; each filter, mapper, and reducer is usually specified as an inner class, so simple queries like "find the highest GPA of any student graduating this year" is still on the order of a dozen lines of code. One of the possibilities for addition to the Java language in Java 7 is closures;

one of the arguments in favor of closures is that it makes expressing small snippets of code — such as filters, mappers, and reducers in `ParallelArray`— much more compact.

Listing 4 shows the max-GPA query rewritten using the BGGA closures proposal. (In the version of `ParallelArray` extended with function types, the type of the parameter to `withFilter()` is not `Ops.Predicate<T>`, but instead the function type `{ T => boolean }`.) The closures notation strips away the boilerplate associated with inner classes, allowing a more compact (and more importantly, more direct) expression of the desired data operation. Now the code is down to three lines, almost all of it expressing some important aspect of the result we are trying to achieve.

Listing 4. Writing the max-GPA example using closures

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

Summary

As the number of available processors increases, we're going to need to find finer-grained sources of parallelism in our programs. One of the most attractive candidates for doing so is aggregate data operations — sorting, searching, and summarizing. The fork-join library, to be introduced in JDK 7, offers a means of "portably expressing" a certain class of parallelizable algorithm, so that the program can run efficiently on a range of hardware platforms. The `ParallelArray` component of the fork-join library can make it even easier to express parallel aggregate operations by declaratively describing the operation you want to perform and letting `ParallelArray` figure out how to execute it efficiently.

Related topics

- "Java theory and practice: Stick a fork in it, Part 1" (developerWorks, November 2007): Explore how fork-join provides a natural mechanism for decomposing many algorithms to effectively exploit hardware parallelism.
- 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 Web site: Download the fork-join framework as part of the jsr166y package, or read the paper on its design.