Java Streams, Part 4: From concurrent to parallel

Understanding the factors influencing parallel performance

Brian Goetz

July 18, 2016

This fourth installment of the Java Streams series identifies and explains factors that determine the effectiveness of parallel processing, putting them into historical and technical context. An understanding of these factors provides a foundation for making optimal use of the Streams library for parallel execution. (The next installment applies the principles outlined here directly to Streams.)

View more content in this series

This fourth installment of the *Java Streams* series explains factors that determine the effectiveness of parallel processing, putting them into historical and technical context. An understanding of these factors provides a foundation for making optimal use of the Streams library for parallel execution — and the next installment focuses on applying these principles directly to Streams.

More cores, not faster cores

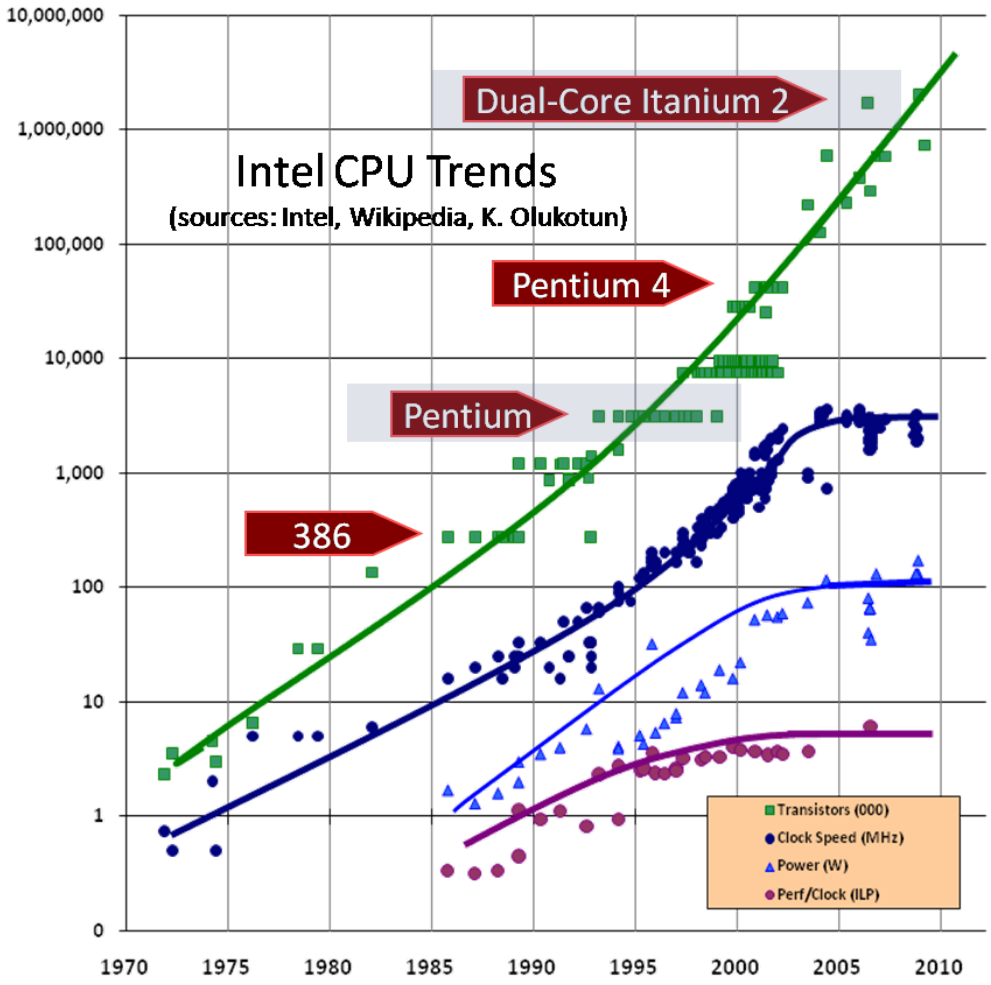
By around 2002, the techniques that chip designers had been using to deliver exponentially increasing performance had started to dry up. Increasing clock rates much further was impractical for various reasons, including power consumption and heat dissipation, and the techniques for doing more work per cycle (*instruction-level parallelism*) had also started to hit the point of diminishing returns.

About this series
With the `java.util.stream` package, you can concisely and declaratively express possibly-parallel bulk operations on collections, arrays, and other data sources. In this series by Java Language Architect Brian Goetz, get a comprehensive understanding of the Streams library and learn how to use it to best advantage.

Moore's law predicts that the number of transistors that can be packed onto a die doubles approximately every two years. When chip designers hit the *frequency wall* in 2002, it wasn't because Moore's Law had run out of steam; we continued to see steady exponential growth in transistor counts. But while the ability to harness this ever-increasing transistor budget into **faster cores** had played out, chip designers could still use this increasing transistor budget to put **more cores** on a single die. Figure 1 illustrates this trend with data from Intel processors, plotted on a

log scale. The topmost (straight) line indicates exponential growth in transistor count, while the lines representing clock rate, power dissipation, and instruction-level parallelism all show a clear leveling off around 2002.

Figure 1. Transistor counts and CPU performance for Intel CPUs (image source: Herb Sutter)



Having more cores might enable greater power efficiency (cores not being actively used can be powered down independently), but it doesn't necessarily equate to better program performance — unless you can keep all the cores busy with useful work. Indeed, today's chips are not giving us as many cores as Moore's law would allow — largely because today's software cannot exploit them cost-effectively.

From concurrent to parallel

Throughout most of the the history of computing, the goal of concurrency has largely stayed the same — improve performance by increasing CPU utilization — but the techniques (and measure of performance) have changed. In the days of single-core systems, concurrency was mostly about *asynchrony*— allowing an activity to relinquish the CPU while waiting for an I/O to complete. Asynchrony could improve both *responsiveness* (not freezing the UI while a background activity is in progress) and *throughput* (allowing another activity to use the CPU while waiting for the I/O to complete). In some concurrency models (Actors and Communicating Sequential Processes [CSP], for example) the concurrency model contributed to the structure of the program, but for the most part (for better or worse) we just used concurrency as needed for performance.

> *“ Factors that influence the effectiveness of parallelism include the problem itself, the algorithm used to solve the problem, the runtime support for task decomposition and scheduling, and the size and memory locality of the data set. ”*

As we moved into the multicore era, the primary application of concurrency was to factor the workload into independent, coarse-grained tasks — such as user requests — with the aim of increasing throughput by processing multiple requests simultaneously. Around this time, the Java libraries acquired tools such as thread pools, semaphores, and futures, which are well suited to task-based concurrency.

As core counts continue to increase, however, enough "naturally occurring" tasks might not be available to keep all the cores busy. With more cores available than tasks, another performance goal becomes attractive: reducing *latency* by using multiple cores to complete a single task more quickly. Not all tasks are easily amenable to this sort of decomposition; the ones that work best are data-intensive queries wherein the same operations are done over a large data set.

Sadly, the terms *concurrency* and *parallelism* don't have standard definitions, and they are often (erroneously) used interchangeably. Historically, *concurrency* described a property of a **program**— the degree to which a program is structured as the interaction of cooperating computational activities — whereas *parallelism* was a property of a program's **execution**, describing the degree to which things actually happen simultaneously. (Under this definition, concurrency is the **potential** for parallelism.) This distinction was useful when true concurrent execution was mostly a theoretical concern, but it has become less useful over time.

More modern curricula describe *concurrency* as being about correctly and efficiently controlling access to shared resources, whereas *parallelism* is about using more resources to solve a problem faster. Constructing thread-safe data structures is the domain of concurrency, as enabled by primitives such as locks, events, semaphores, coroutines, or software transactional memory (STM). On the other hand, parallelism uses techniques like partitioning or sharding to enable multiple activities to make progress on the task **without** coordination.

Why is this distinction important? After all, concurrency and parallelism have the common goal of getting multiple things done simultaneously. But there's a big difference in how easy the two are to get right. Making concurrent code correct with coordination primitives such as locks is difficult, error prone, and unnatural. Making parallel code correct by arranging that each worker has its own portion of the problem to work on is comparatively simpler, safer, and more natural.

Parallelism

While the mechanics of parallelism are often straightforward, the hard part is knowing **when** to use it. Parallelism is strictly an optimization; it is a choice to use more resources for a particular computation, in the hope of getting to the answer faster — but you always have the option of performing the computation sequentially. Unfortunately, using more resources is no guarantee of getting to the answer faster — or even as fast. And, if parallelism offers us no return (or negative return) on additional resource consumption, we shouldn't use it. Of course, the return is highly situational, so there's no universal answer. But we have tools to help us evaluate if parallelism will be effective in a given situation: analysis, measurement, and performance requirements.

This article focuses mostly on analysis — exploring which kinds of computations tend to parallelize well and which don't. As a rule of thumb, though, prefer using a sequential execution unless you have reason to believe that you will get a speedup from parallelism, **and** the speedup actually matters according to your performance requirements. (Many programs are already fast enough, so effort spent optimizing them could be better spent on more value-creating activities, such as improving usability or reliability.)

The measure of parallel effectiveness, called *speedup*, is simply the ratio of parallel runtime to sequential runtime. Choosing parallelism (assuming it delivers a speedup) is a deliberate choice to value time over CPU and power utilization. A parallel execution always does more work than a sequential one, since — in addition to solving the problem — it also must decompose the problem, create and manage tasks to describe the subproblems, dispatch and wait for those tasks, and merge their results. So the parallel execution always starts out "behind" the sequential one and hopes to make up for the initial deficit through economy of scale.

For parallelism to be the better choice, several things must come together. We need a problem that admits a parallel solution in the first place — which not all problems do. Then, we need an implementation of the solution that exploits the inherent parallelism. We need to ensure that the techniques used to implement parallelism don't come with so much overhead that we squander the cycles we throw at the problem. And we need **enough data** so that we can achieve the economy of scale needed to get a speedup.

Exploiting parallelism

Not all problems are equally amenable to parallelization. Consider the following problem: Given a function `f`, which we'll assume is expensive to compute, define `g` as follows:

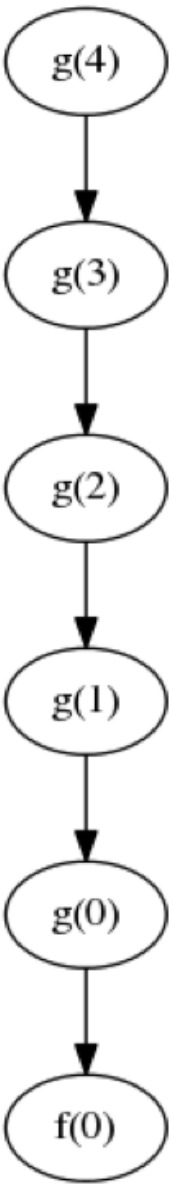
```
g(0) = f(0)
g(n) = f( g(n-1) ), for n > 0
```

We could implement a parallel algorithm for `g` and measure its speedup, but we don't need to; we can look at the problem and immediately see that it is fundamentally sequential. To see this, we can rewrite `g(n)` slightly differently:

```
g(n) = f( f( ... n times ... f(0) ... )
```

With this rewrite, we see that we can't even start computing `g(n)` until we finish computing `g(n-1)`. In the dataflow dependency diagram for computing `g(4)`, shown in Figure 2, each value of `g(n)` depends on the previous value `g(n-1)`.

Figure 2. Dataflow dependency graph for function g

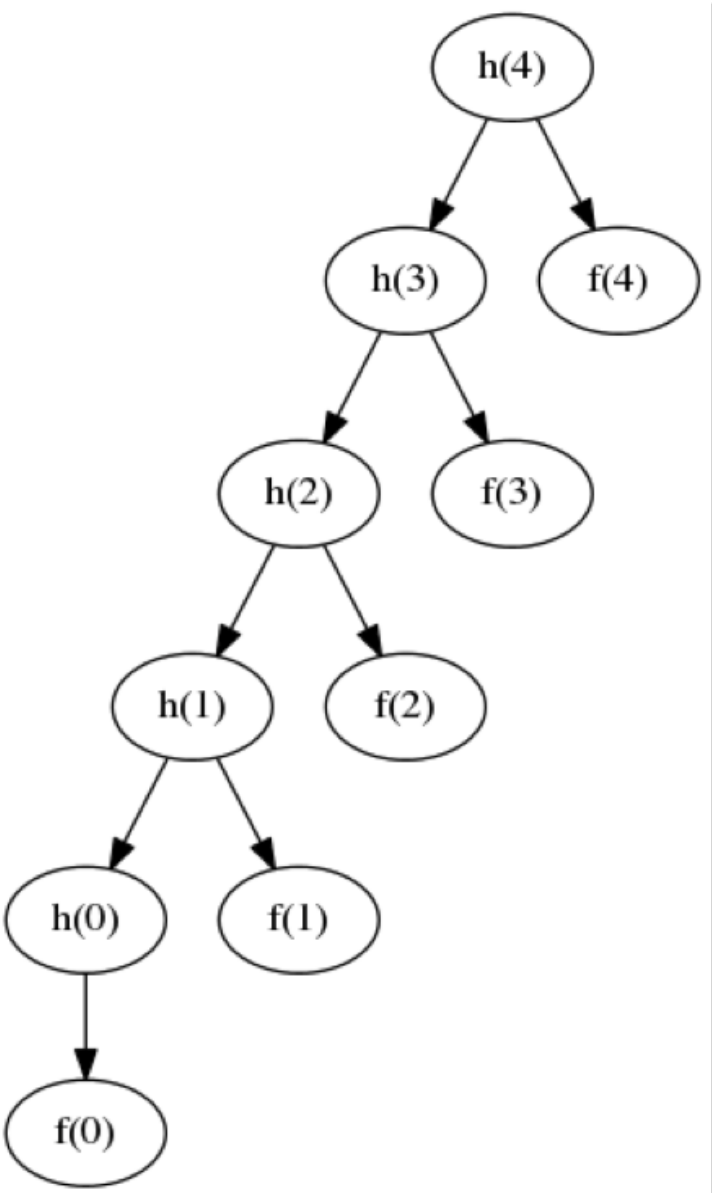


You might be tempted to conclude that the problem stems from the fact that `g(n)` is defined recursively, but that's not the case. Consider the slightly different problem of computing the function `h(n)`:

```
h(0) = f(0)
h(n) = f(n) + h(n-1)
```

If we write the obvious implementation for computing `h(n)`, we end up with a dataflow dependency graph like the one shown in Figure 3, in which each `h(n)` depends on `h(n-1)`.

Figure 3. Dataflow dependency graph for a naive implementation of function h

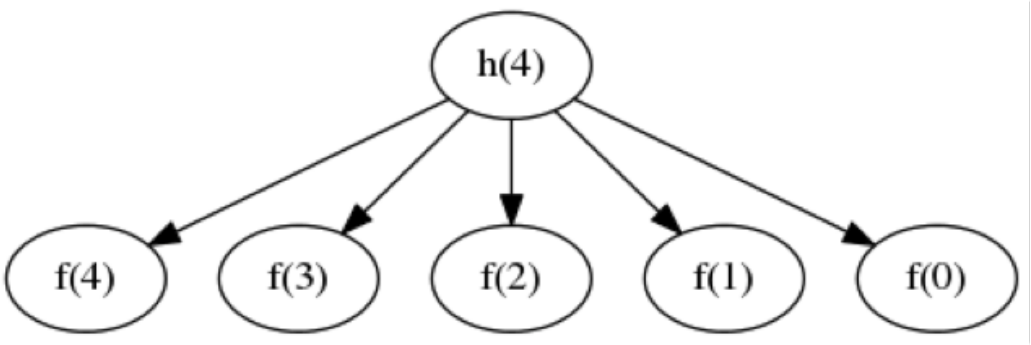


However, if we rewrite `h(n)` in a different way, we can immediately see how this problem has exploitable parallelism. We can compute each of the terms independently and then add them up (which also admits parallelism):

```
h(n) = f(0) + f(1) + .. + f(n)
```

The result is a dataflow dependency graph like the one shown in Figure 4, in which each `h(n)` can be computed independently.

Figure 4. Dataflow dependency graph for function h



These examples show us two things: first, that similar-looking problems can have vastly different degrees of exploitable parallelism; and second, that the "obvious" implementation of a solution to a problem with exploitable parallelism might not necessarily **exploit** that parallelism. To have any chance of getting a speedup, we need both.

Divide and conquer

The standard technique for attacking exploitable parallelism is called *recursive decomposition*, or *divide and conquer*. In this approach, we repeatedly divide the problem into subproblems until the subproblems are small enough that it makes more sense to solve them sequentially; we then solve the subproblems in parallel, and combine the partial results of the subproblems to derive a total result.

Listing 1 illustrates a typical divide-and-conquer solution in pseudocode, using a hypothetical `CONCURRENT` primitive for concurrent execution.

Listing 1. Recursive decomposition

```
R solve(Problem<R> problem) {
    if (problem.isSmall())
        return problem.solveSequentially();
    R leftResult, rightResult;
    CONCURRENT {
        leftResult = solve(problem.leftHalf());
        rightResult = solve(problem.rightHalf());
    }
    return problem.combine(leftResult, rightResult);
}
```

Recursive decomposition is appealing because it is **simple** (especially so when dealing with data structures that are already defined recursively, such as trees). Parallel code like that in Listing 1 is portable across a range of computing environments; it works efficiently with one core or with many, and it need not concern itself with the complexities of coordinating access to shared mutable state — because there is no shared mutable state. Partitioning the problem into subproblems, and

arranging for each task to only access the data from a particular subproblem, generally requires no coordination between threads.

Performance considerations

With Listing 1 as our model, we can now proceed to analyze the conditions under which parallelism could offer an advantage. Two additional algorithmic steps are introduced by divide-and-conquer — dividing the problem, and combining the partial results — and each of these can be more or less friendly to parallelism. Another factor that can affect parallel performance is the efficiency of the parallelism primitives themselves — illustrated by the hypothetical `CONCURRENT` mechanism in Listing 1's pseudocode. An additional two factors are properties of the data set: its size and its memory locality. We consider each of these conditions in turn.

Some problems, such as the `g(n)` function in the "Exploiting parallelism" section, do not admit decomposition at all. But even when a problem admits decomposition, decomposing it can have a cost. (At the very least, decomposing a problem involves creating a description of the subproblem.) For example, the decomposition step of the Quicksort algorithm requires finding a pivot point, which is *O(n)* in the size of the problem, because it involves examining and possibly updating all the data. This requirement is much more expensive than the partitioning step of a problem like "find the sum of an array of elements," whose partitioning step is *O(1)*— "find the average of the top and bottom indices." And, even if the problem can be efficiently decomposed, if the two subproblems are of vastly unequal size, we might not have exposed much exploitable parallelism.

Similarly, when we solve two subproblems, we must combine the results. If our problem is "remove duplicate elements," the combination step requires the merging of two sets; if we want a flat representation of the result, this step is also *O(n)*. On the other hand, if our problem is "find the sum of an array," our combination step is again *O(1)*— "add two numbers."

Managing tasks to be executed concurrently can involve several possible sources of efficiency loss, such as the inherent latency of handing off data from one thread to another or the risk of contention for shared data structures. The *fork-join* framework — added in Java SE 7 for managing fine-grained, computation-intensive tasks — is designed to minimize many common sources of inefficiency in parallel dispatch. The `java.util.stream` library uses the fork-join framework for implementing parallel execution.

Finally, we must consider the data. If the data set is small, it will be hard to extract any sort of speedup, because of the startup costs imposed by parallel execution. Similarly, if the data set exhibits poor memory locality (pointer-rich data structures such as graphs, as opposed to arrays), it is likely that — with today's typically memory-bound systems — executing in parallel will simply result in many threads waiting for data from the cache, rather than effectively using the cores to compute the answer faster.

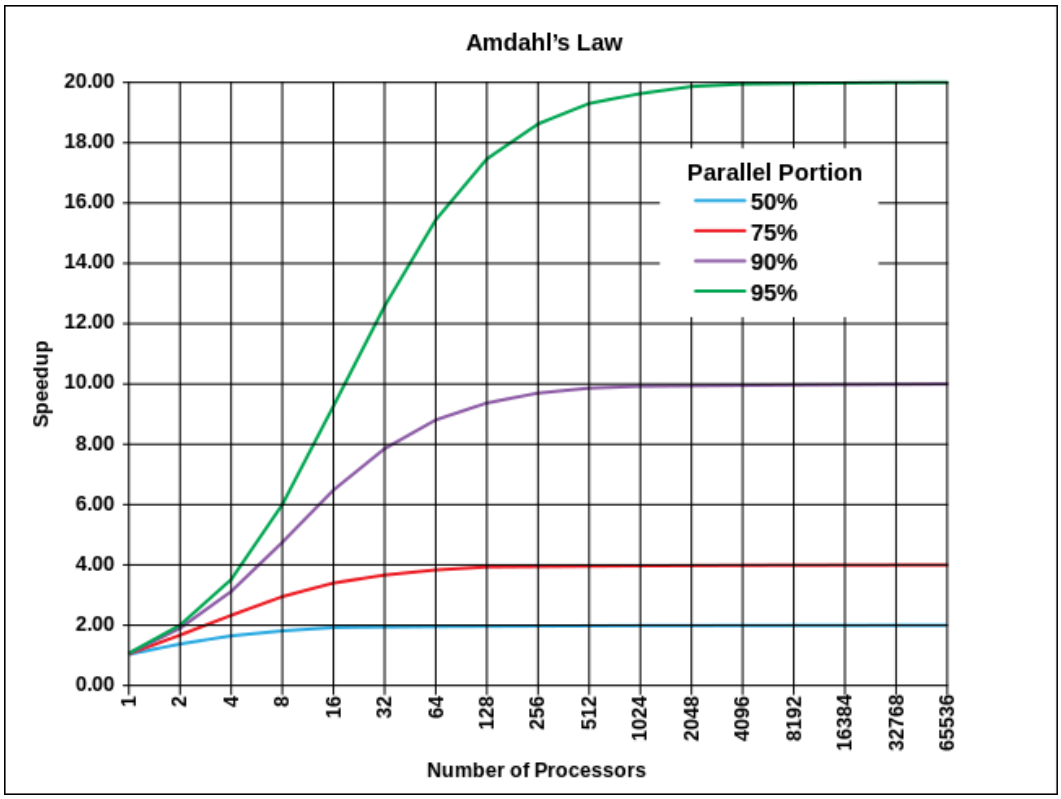
Each of these factors can steal from our speedup; in some situations, they can not only erase the speedup but even cause a slowdown.

Amdahl's law

Amdahl's law describes how the sequential portion of a computation limits the possible parallel speedup. Most problems have some amount of work that cannot be parallelized; this is called the *serial fraction*. For example, if you are going to copy the data from one array to another, the copying might be parallelizable, but the allocation of the target array — which is inherently sequential — must happen before any copying can happen.

Figure 5 shows the effects of Amdahl's law. The various curves illustrate the best possible speedup allowed by Amdahl's Law as a function of the number of processors available, for varying parallel fractions (the complement of the serial fraction.) If, for example, the parallel fraction is .5 (half the problem must be executed sequentially) and an infinite number of processors are available, Amdahl's law tells us that the best speedup we can hope for is 2x. This is intuitively sensible; even if we could reduce the cost of the parallelizable part to zero through parallelization, we still have half the problem to solve sequentially.

Figure 5. Amdahl's law (image source: Wikimedia Commons)



The model implied by Amdahl's Law — that some fraction of the work must be completely sequential and the rest is perfectly parallelizable — is overly simplistic. Nonetheless, it is still useful for understanding the factors that inhibit parallelism. The ability to spot, and reduce, the serial fraction is a key factor in being able to craft more efficient parallel algorithms.

Another way to interpret Amdahl's law is: If you have a 32-core machine, for every cycle you spend setting up a parallel computation, that's 31 cycles that can't be applied to solving the problem. And

if you've split the problem in two, that's still 30 cycles going to waste on every clock tick. Only when you can split off enough work to keep all the processors busy are you fully up and running — and if you don't get there quickly enough (or stay there long enough), it's going to be hard to get a good speedup.

Conclusion to Part 4

Parallelism is a trade-off of using more compute resources in the hope of getting a speedup. While in theory we can speed up a problem by a factor of *N* by using *N* cores, reality usually falls far short of this goal. Factors that influence the effectiveness of parallelism include the problem itself, the algorithm used to solve the problem, the runtime support for task decomposition and scheduling, and the size and memory locality of the data set. Part 5 of *Java Streams* applies these considerations to the Streams library and shows how (and why) some stream pipelines parallelize better than others.

Related topics

- The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software
- Moore's law
- Amdahl's law
- Java Concurrency in Practice (Brian Goetz, et al., Addison-Wesley Professional)
- Package documentation for java.util.stream
- The Java fork-join library
- IBM Code: Java journeys