11-2013

What's in an Algorithm?

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Recommended Citation

G. K. Thiruvathukal, "What's in an Algorithm?," in Computing in Science & Engineering, vol. 15, no. 4, pp. 4-5, July-Aug. 2013. doi: 10.1109/MCSE.2013.94

What's in an Algorithm?

By George K. Thiruvathukal

THE FUTURE OF HIGH-PERFORMANCE COMPUTING'S (HPC'S) SUCCESS MIGHT WELL DEPEND ON BEING ABLE TO *THINK* IN PARALLEL, STARTING WITH THE DEFINITION OF AN ALGORITHM:

> In mathematics and computer science, an algorithm is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning.
>
> —*Wikipedia*

> A procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end especially by a computer.
>
> —*Merriam-Webster*

At a recent IEEE Computer Society Board of Governors meeting, I was invited by Jean-Luc Gaudiot to join several members of the Educational Activities Board (and other colleagues) for dinner. Jean-Luc was particularly keen on knowing my thoughts on how I "define" an algorithm. It turns out that the two of us have something in common (research in dataflow computing), which might explain why he wanted to know my definition. I'll discuss this more later.

Being Enlightened

First, we need a bit of background. Long-time readers of the "Scientific Programming" department will know that I have a longstanding interest in exploring alternatives to the imperative style of programming that still predominates today's "computational thinking" landscape. In the late 1980s and early 1990s, I was working on the implementation of functional/dataflow languages that could compile and execute on supercomputers and networks of workstations (the early term for clusters). Although these early efforts were promising, I could never quite make the case that working in functional and object-oriented languages was the right thing to do. Thinking functionally is tricky, owing to our classically conditioned learning of procedural programming (in the von Neumann architectural style). This ultimately led me to return to more hybrid (better read as "conservative") approaches based on C/C++ (and Fortran), which have been carrying the torch of HPC for some time now. And it's clear that they'll continue to play a role in the modern era.

This background information is important, because it has much to do with how an algorithm is "defined." The term algorithm is largely credited to Al-Khwarizmi of Baghdad, who with his contemporaries in the House of Wisdom introduced the notion of calculation using Hindu numerals to the Western world during a period often called the Dark Ages (in Europe) that preceded the age of enlightenment. Without this important contribution to mathematics and computational thinking, it's hard to imagine being enlightened.

While it would be some time before these ideas actually gained traction (until well into the early 1900s), a definition of algorithms ultimately ensued. In both the *Wikipedia* and *Merriam-Webster* definitions, the phrasing of *step-by-step* and *procedure* are practically joined at the hips. There's nothing technically wrong with this definition, but it shapes our thinking about algorithms in a way that could hamper effective teaching of computational and computer science disciplines, not to mention limiting our imagination when it comes to solving a problem in a way that lends itself to execution on modern computer hardware (such as supercomputers, accelerators, and other novel architectures).

Removing Limitations by Thinking Functionally

Let's start with the thinking that accompanies step-by-step. Again, we know from Turing computability that all computation can be boiled down to a sequence of state transitions and actions, so steps are at least innate to modern computing hardware. But does the actual thinking behind an algorithm really require us to use steps (or imperative statements)? The answer seemingly points in the negative. One of the first algorithms I learned in computing (even before becoming a computer scientist) was in my discrete mathematics course: the greatest common divisor,

 Copublished by the IEEE CS and the AIP 1521-9615/13/$31.00 © 2013 IEEE

the definition and solution of which is credited to Euclid (another pre-Enlightenment and even pre-Dark Ages fellow). Its definition is recursive:

```
gcd(a, b) = gcd(b, a mod b)
        gcd(a, 0) = a
    where a >= 0, b > 0
```

I'm going to point to my lecture notes (http://introcs.cs.luc.edu/book/latest/html/default/gcdexamples.html) for the details and various ways of coding it (good and bad, followed by better). The lowdown is that this elementary algorithm shows how computational thinking (which, in part, includes mathematical thinking) doesn't require us to express a computation as a series of steps. Yes, the code might ultimately be compiled into a series of steps, but in this case the algorithm is expressed as a recursive composition of functions.

And this isn't the only algorithm of its kind. Another famous example is the sorting algorithm quicksort (also covered in our lectures) and the Strassen algorithm for matrix multiplication (one that many of our readers know). All of these algorithms are conveniently expressed using recursive function composition. You'll ultimately find yourself writing the code as "steps" in languages such as C/C++ and Fortran, but it's actually possible to write the code without steps in modern functional programming languages.

So why should we care? Well, as it turns out, we're seeing a resurgence of interest in alternative paradigms for parallel computing (something I hope we'll do a special issue on next year). We've done a special issue on Modern Programming Languages, wherein the authors demonstrated how an example algorithm can be coded in a functional style—recursive and without side effects (read: global variables). When we think about programs in a functional way without side effects and a priori serialization, we have more opportunities to exploit multicore and emerging architectures. That's because any function *application* (that is, a function call) is a natural candidate for parallelization. This can lead to a superfluity of parallel computation, but it can be managed using pooled thread execution or, better yet, actors that don't require the full resources of a thread (that is, no lock variables or thread state).

So does this answer Jean-Luc's question? I hope so. In the end, teachers, researchers, and practitioners should update the algorithm's definition. With parallel (and distributed) computers now on everyone's desktop, phones, and gaming systems, I posit that a culture of reliability and reproducibility needs to be part of what we're doing. Functional computing was created by mathematicians/logicians for mathematicians/logicians. It gives me pause to think about what the world would look like if Al-Khwarizmi's efforts to bring the Hindu system to the West hadn't succeeded. We probably wouldn't be working with hieroglyphics or Roman numerals, but we wouldn't be harnessing the full power of computation, either.

Today, the analog of this for computer/computational scientists is to be looking at the immense promise—and delivery—of modern functional programming languages, which are not just toy languages anymore. As the EIC of *CiSE*, I promise to bring more of this content to you with the hope that it can be put into practice for solving the world's most important computational/engineering problems.