Parallel Computational Thinking

By Keith Kirkpatrick
Communications of the ACM, Vol. 60 No. 12, Pages 17-19
10.1145/3148760
Comments (2)
Credit: Getty Images

When learning a new skill, it is often advantageous to start out simply, and then incorporate greater complexity as the learner gains greater experience, expertise, and familiarity with the subject at hand.

Indeed, most computer science education has followed that line of thinking, teaching beginning computer science students to write programs that perform one instruction at a time, and then move on to the next instruction. This is known as sequential programming, and it has largely been the accepted model of computer science instruction at both the university and K–12 levels, in contrast with parallel computing, a model of programming where multiple instructions are processed simultaneously.

"The educational system is, mostly through inertia, still focused on the computing paradigm of the 20th century, which was one processor executing instructions one after another, so algorithmic problem-solving was mainly oriented toward a sequential model," explains Charles (Chip) Weems, an associate professor of computer science at the University of Massachusetts.

Today, however, nearly all applications running on smartphones, tablets, and PCs, are powered by multi-core processors, which are necessary when working with the large datasets that drive both consumer applications, such a Twitter or Facebook feed, as well as business and commerce-related applications, such as travel deal sites, weather applications, and real-time traffic data. To take full advantage of these multicore processors, programming applications to process instructions in parallel—which allows multiple instructions to be processed at the same time—is required.

Teaching new computer science students to think and program in parallel will not only better prepare them to code and program these devices, but also helps to train their minds to think in abstractions to solve problems, rather than simply in terms of writing code.

"Computer architectures have been doing parallelism at the instruction level for decades in a way that the vast majority of programmers can ignore that it's there," says Dan Grossman, a professor at the Paul G. Allen School of Computer Science & Engineering at

the University of Washington, and a member of the ACM steering committee on computing curricula, which concluded its work in 2013. "If that had remained the only form of parallelism, there would be a much weaker argument for teaching parallelism at the undergraduate level. But that has not remained the dominant form; it has run its course. We have added other forms of parallelism that do not hide the issue as much for programmers."

Computing societies have recognized the need to incorporate parallelism as part of a core collegiate computer science curriculum. The ACM and the IEEE jointly introduced new guidelines in 2013, and recommended integrating parallel education throughout the curriculum. Although these are only guidelines, and most universities still tend to teach parallel programming concepts only to more advanced students, there is a growing push to incorporate parallelism in college-level programming courses from the start.

"In the last 15 years, systems have gone almost entirely parallel," Weems says. "Unless you're talking about small embedded systems, everything you might encounter is a multicore that's multithreaded, and nearly everything comes with a graphics processor, which can be programmed."

Moreover, programming in parallel—and training students to think at a higher level—allows for programming to be more direct and concise, compared with sequential programming, according to Guy Blelloch, a professor and Associate Dean for Undergraduate Programs in the computer science department at Carnegie Mellon University. "It's not so much that parallel code is simpler than sequential code, it's that in the abstraction of code [it] basically makes [the problem to solved] simpler and at the same time makes it parallel."

Blelloch, who notes that CMU teaches parallelism from the start in its Intro to Data Structures and Algorithms Course during the first semester of sophomore year, says that by teaching beginning computer science students to think about problems in terms of abstractions, they are able to move beyond a basic understanding of programming to get the heart of the matter of actually solving problems.

"There's a lot of emphasis on intro programming on a loop," Blelloch says. "And that's really not that interesting. Often you just want to be thinking, 'I want to add five to every element in this array.' I could start the loop at the beginning, but the right way to think of it is I just want to add five to every element in the array. By doing the parallelism, you're more focusing on the underlying ideas, as opposed to getting stuck in details of loops."

Another added benefit of parallelism, Weems says, is that when a student learns parallel programming, it "helps them develop a more flexible approach to problem solving because there are more algorithmic models to draw upon." While there's also the added benefit of learning how to break apart larger problems into simpler ones, parallelism also requires programmers to learn to see alternate abstractions of the problem.

"There are situations where a problem can be decomposed into subtasks, but various factors result in still having to choose among algorithmic approaches," Weems says. "There are times when communicating via shared memory is most effective, while in other cases it's better to work locally and communicate via messages, or to use a combination of these approaches at different levels of granularity. [Parallelism] forces programmers to look

more explicitly and holistically at the interactions that take place among the data and operations in solving the problem, by considering them from more perspectives than the sequential model."

Weems, a member of the working group for the Center for Parallel and Distributed Computing Curriculum Development and Educational Resources (CDER), which is funded by the U.S. National Science Foundation (NSF), also notes that there is a demand for new programmers who have parallel programming skills, from government science labs as well as large technology industry companies.

Some major universities, including the University of Massachusetts, where Weems is a faculty member, have begun to incorporate parallelism into their curricula, with promising results. For example, a faculty-authored paper from Texas State University highlights the success of its new curriculum, which was launched in the 2016–2017 academic year. According to the paper, parallel computing concepts are introduced and reiterated via a series of short, self-contained modules across several lower division courses. Then, most concepts are combined into a senior-level capstone course in multicore programming. The evaluations conducted during the first year displayed encouraging results for the early-and-often approach in terms of learning outcomes, student interest, and confidence gains in computer science.

Still, some educators are not convinced that introducing parallelism during introductory or lower-division computer science courses is necessary in order to produce well-trained computer programmers of the future.

"We live in a world of multicore devices," says Mark Guzdial, a professor in the School of Interactive Computing at the Georgia Institute of Technology. "But I don't know if we should be teaching [parallel programming] to everyone. It may be better off to start with sequential programming, and then move on to parallel."

Whether to teach elements of parallelism in early coursework may also depend on the focus of the coursework, Grossman says, noting that the key to integrating parallelism is to limit the complexity of the program itself, by ensuring that computations and variables are not highly dependent upon one another during parallel processing operations.

"The way to get parallelism to work correctly is to have as fewer shared variables as you can that can change," Grossman says, noting instead of setting up a single variable that may change its value, it may make more sense to simply program a second variable that can hold the second value.

"There are different ways to teach introductory programming without parallelism that make it harder or easier to add parallelism later," Grossman adds. "For initial exposure to programming for younger, pre-college students, I haven't seen much focus on parallelism and I think that's fine."

Hansel Lynn owns Silicon Valley-based theCoderSchool, an afterschool coding instruction franchise that works exclusively with children ages 8 to 18. Lynn believes sequential coding should be taught first. "For kids aged 8–18, we always teach sequential coding first," Lynn

says. "Especially for younger kids, sequential, logical stepping is an important foundation for learning how to think logically."

Lynn does note that some platforms may provide some exposure to parallel programming. "At times however, platforms such as Scratch do allow kids to get some exposure to parallel programming (such as multiple objects moving and detecting collision simultaneously)," Lynn says. "While we may create projects with parallel programming concepts embedded, we don't typically focus on parallel thinking, as we find it dilutes the focus on building their sequential logic thinking skills."

Whether at the collegiate or secondary level, there are challenges related to revamping the curriculum to include parallelism. First, many professors have not had been exposed to parallelism on a programming level, particularly if they were educated before parallel processing became mainstream, which occurred about a decade or so ago. There are also non-technical issues, such as getting buy-in from other faculty members, as well as the challenge of updating online tutorials and auto-graders, which must be revamped to deal with different types of code. Additionally, textbooks need to be augmented or amended, as many introductory texts don't cover parallelism. "There was one that mentioned concurrency, but it was in Chapter 23," Weems quips.

Blelloch says that the trend, for now, is to simply add a discussion about parallelism to existing coursework. "I think most departments are taking a some-what more conservative approach," he says. "They've taught this course for 20 years in a particular way and it's not very hard to add three weeks at the end which makes them think in parallel."

Perhaps the larger question for computer science educators revolves around selecting the right material to introduce to beginning computer science students.

"It's very tempting in computer science education to think that we do students a service if we introduce 'x' from day one, for various values of 'x', and you're asking about 'x' being parallelism," Grossman says. "I do see value in that, but I also see value in introducing security from day one. I see value in introducing ethics from day one. I also see the value of introducing performance from day one. But there's only one day one.

"Everyone who crafts a curriculum has to make choices about what they introduce from the beginning as the default way to think about a program, compared with what they push off until later," Grossman says. "Trade-offs are trade-offs, and people can spend their lives studying pedagogy."

Author

Keith Kirkpatrick is principal of 4K Research & Consulting, LLC, based in Lynbrook, NY.

Comments

George Rudolph

November 28, 2017 01:03

It is most often useful, in advanced courses, to have students write a sequential version and then a parallel version. One reason is to see the performance differences, and one reason is it helps verify that the parallel version is going to work correctly.

I am a firm believer that CS curriculum needs to change, and that the change starts first with changing how we think about computation, and what professors/instructors are comfortable teaching. I agree with what the author has said here.

Keith Kirkpatrick

November 28, 2017 02:37

George--thanks for reading and for your comment. Computer science, like many other disciplines that focus on real-world skills, must continue to evolve to keep pace with the changes in the industry.