Don't Panic!

A position on curricula for concurrency

Stephen Kell
Stephen.Kell@cl.cam.ac.uk

Abstract

Concurrency is a pervasive concept and should be taught as such. However, concurrent programming is intrinsically hard, and knowing how and when to avoid it is as important as knowing how to do it.

1 Introduction

I'm a graduate student and have taught second-year undergraduates in small-group sessions on concurrent programming for four years now. I enjoy teaching concurrent programming because I find it hard. Every year my own understanding is improved thanks to my teaching work. To become skilled takes experience, as with all programming work, and in particular because experience confers two things: conceptual clarity and judgement. In a nutshell, my position is that curricula must focus on accelerating students' progress to these two goals. We must emphasise judgement and rationale over technique and technology. Concurrent programming, in all its guises, represents a trade-off, just as sequential programming does. The latter is unlikely to go away, because sequential thinking comes naturally to humans. Just as with all programming tasks, we must teach students how to understand the underlying trade-offs, know when various styles and techniques are suitable, and know *how to evaluate* the suitability of techniques both present and future. My "nightmare scenario" is that in training a generation of programmers to unthinkingly use concurrent programming practices, we will yield a huge quantity of unreliable and unmaintainable code, whose costs far outweigh any parallel performance gains.

2 Spending the transistor count

Throughout the 1980s and 1990s, an age of skyrocketing sequential performance, software generally didn't get much faster. Niklaus Wirth famously observed this in his "plea for lean software" in the early 1990s [3]. For sure, certain new application classes opened up—commodity computers became capable of astonishing feats of multimedia processing in software, for example. But in general, the popular everyday applications of the last fifteen years (e-mail, the web, word processing, and so on) have not been revolutionised by ongoing increases in transistor counts.

If anything, transistors have effectively been spent in enabling programmers to deliver results more quickly—spending less time on optimisation, using more profligate abstractions and coding styles, and pressing existing software into use even in cases where it is inefficient, so as to avoid the effort of creating and proving a new system.[1] Recent developments continue this trend: a huge number of applications are now delivered as interpreted scripts (JavaScript), transmitted in inefficient plain-text encodings and storing their data similarly inefficiently (as XML), executing on poorly optimised runtimes (web browsers). In the meantime, transistors been spent on making programmers' lives easier and on hastening the social processes of software: the web has allowed socially useful new deployment models, driven by technology which is rather ill suited but near-at-hand (namely XML, which happened to derive from publishing systems, and JavaScript, whose original role was far narrower than its use today). This has avoided the time-consuming extra step of developing and standardising technologies which are optimised for these purposes (which could be far more efficient, but would cost considerable time and effort to produce).[2]

If we believe that increased transistor counts, now being spent on multiple cores, are to be used for running highly concurrent programs, then it's likely we believe that this story is to be reversed: programmers' lives are to get a lot harder, for the sake of exploiting these processors. I would find this a strange development. Earlier this year when marking some exercises for my students, I discovered a major bug in the model answer for a past exam question, which had been written by the lecturer several years ago and had stood uncorrected since. We should not underestimate how intrinsically hard concurrency is: if lecturers and graduate students make these errors, then however much we improve pedagogy, educating a generation of professional programmers who *don't* make them seems like an impossible task.

(There is a chance that, thanks to very clever runtime support, programmers will be able to write highly parallel code without worrying about most of the complexity of concurrency. This is an ideal situation, but it seems unlikely to me. I would greatly value the opinion of other workshop participants on this, and on the related question of whether our curricula are intended primarily to educate the experts who build the very clever runtimes, or those who merely use them. I am

[1] Butler Lampson observed this in his essay "Software Components: Only the Giants Survive" [1], although in light of my disagreement with the remainder of piece, this should not be considered a positive citation.

[2] It's arguable as to whether this has optimised the value of our software, but that's a separate question.

writing on the assumption that the primary consideration is the latter category; the former class probably begin as members of the latter.)

3 Joining the dots

It might seem at this stage that I am advocating something ridiculous, such as a reduction or wholesale abandonment of concurrency teaching. On the contrary, the importance of concurrency cannot be understated. Concurrency is a pervasive concept: in programming, in computers and in systems generally. Accordingly, it is not new. Long before concurrent programming became popular, we have had inter-device concurrency, multiprogramming, asynchronous programming interfaces, reentrancy, coroutines and so forth. One of the easy failings in teaching computer science is neglecting to join the dots between kindred concepts appearing in superficially distinct contexts. It therefore seems uncontroversial that concurrency deserves to be treated as a first-class concept whose many guises can be exposed, compared and contrasted in the treatment of the various contexts in which they occur. I therefore fully advocate "sprinkling" concurrency across teaching of various topics—with the view that if those courses didn't mention concurrency previously, there was something wrong with them!

4 Contexts

Algorithms are one major context for discussion of concurrency. Teaching of algorithms is steeped in sequential thinking, and this must change. Sequential algorithms, concurrent algorithms (over a coherent shared store) and fully distributed algorithms deserve equal consideration, and there are huge potential gains from considering these side-by-side rather than, as would seem to be the usual practice, sidelining the non-sequential cases into obscure corners of distributed systems teaching. (As a disclaimer: this may be a quirk of my own institution.)

Language design is another context. When learning threaded programming, one perspective that my students sometimes find useful is to regard threads as a dynamic control structure. Much as a for-loop or if–then–else construct describes what to do and when, so does a set of threads; much as a pointer structure on the heap differs by its dynamism from a set of arrays in static storage, so does a dynamic set of threads from a statically-elaborated control graph. Similarly, it is enlightening to consider how some language constructs are inherently parallelisable (of course threads, but also perhaps a set of subexpressions with no predefined order of evaluation) whereas others are surprisingly sequential (e.g. the sequence points and strictness rules for short-circuit `&&` or `||`). Clearly, language designers consider the trade-offs in deciding what semantics to provide (typically trading off abstraction and comprehensibility with performance); similarly, language users must consider similar trade-offs in the features and styles which they choose to adopt.

5 The skill of restraint

When students first learn about multithreaded programming, it's as if they have been given a new box of toys. Concurrent programming primitives, like threads and locks and condition variables, are intricate and fascinating objects which students are eager to play with. Given any programming exercise, they will almost invariably use threads given any opportunity, and sometimes when given none at all. One exercise my students do is to design a server for a networked tic-tac-toe game. Since it is only ever one player's turn at a time in any game, one thread per game suffices; since games share no data, no locks are needed on the board data structure. None of my students so far has suggested such a design, and rarely has the underlying design process—of determining what degree of parallelism is useful, and identifying what resources are shared—been evident from their answer.

It is too easy to teach courses which describe the mechanisms but not the concepts. My students' designs for the game server mostly do work—but to say a program works is to give it relatively little praise. As Brian Kernighan remarked, since debugging is harder than coding, by writing very smart code you are giving yourself an impossible debugging task. Similarly with threaded programming, one skill is to avoid writing code which is needlessly difficult to understand, debug and maintain. More positively, some excellent tutorial-style articles are available which emphasise concepts over mechanism—Herb Sutter's "Sharing is the Root of all Contention" [2] springs to mind.

6 Introducing concurrency

Should concurrency be an introductory topic? In line with my arguments so far, I'd argue yes—but as a concept, rather than a technique. My institution does well by teaching functional programming (using ML) in its introductory course; this could also be a convenient way of introducing concurrency relatively easily, since parallelisability is clearer and the issues of side-effect, mutable storage etc. already necessarily receive explicit treatment.

References

[1] B. Lampson. Software components: Only the giants survive. In K. Sparck-Jones and A. Herbert, editors, *Computer Systems: Theory, Technology, and Applications*, pages 137–146. Springer, 2004.

[2] H. Sutter. Sharing is the root of all contention. *Dr. Dobb's Journal*, February 2009.

[3] N. Wirth. A plea for lean software. *Computer*, 28(2):64–68, 1995.