# The Impending Ordinariness of Teaching Concurrent Programming

Doug Lea
SUNY, Oswego
dl@cs.oswego.edu

## Abstract

It is a sure thing that topics in concurrency and parallelism will further infiltrate common Computer Science and Software Engineering Curricula. This paper surveys some likely targets, including those on programming techniques, program design, data structures, semantics, and systems.

***Categories and Subject Descriptors*** K.3 Computers and Education [*K.3.2 Computer and Information Science Education*]: Curriculum

***General Terms*** Algorithms, Design, Experimentation, Measurement, Performance, Theory

***Keywords*** Concurrency, Parallelism, Curricula

## 1. Introduction

For the past forty years, Computer Science (and Software Engineering) curriculum design has been largely a matter of compromise in introducing newer ideas while retaining the essence of older, but still improtant ideas. The OOPSLA/Splash community has been (often indirectly) responsible for several curricular shifts over the years. These days, nearly all students are taught the basics of OO programming, UML diagrams, at least a few Gang-of-Four design patterns, and so on. The initial novelty of such topics has long since worn off. For both better and worse, corresponding curricular guidelines focus on ensuring competent coverage of basic principles rather than the excitement associated with introducing incompletely understood or capturing strong differences in opinion among those researching them. For one of many examples, teaching/learning Java vs JavaScript conveys almost nothing about the circa 1990 feuds among class-based vs prototype-based OO language designers.


*SPLASH'10,* October 17–21, 2010, Reno/Tahoe, Nevada, USA.
ACM 978-1-4503-0240-1/10/10.

The same trends are upon us in the realm of concurrent/parallel programming. The general directions of ongoing and upcoming curricular changes are increasingly clear. Some (including those on developing programming skills) seem entirely analogous to those surrounding the introduction of OO programming. Others differ in that concurrency makes contact with a broader set of programming communities as well as work in systems, architecture, and theory. Thus, one might expect more diverse answers to questions including: should concurrency topics be the sole focus of one course (or more), or should ideas be infused across others, or both? The remainder of this paper surveys some of the concrete coverage topics. The intent is not to try to define this coverage, but to expose some likely topics (not courses), to draw attention to those that may benefit from ideas by researchers and practitioners in concurrency. The topics also invite contemplation (without providing answers) of the hard questions of what existing common aspects of the CS curriculum will become reduced or removed.

***Operations on Aggregates.*** Structured (and often "deteministic") parallel programming constructs that perform apply, map, reduce, select, and related operations on the elements of arrays and collections are arguably no harder to learn and use than are sequential for-loops. Which is not to say that either are trivially easy – both admit common error patterns. However, teaching them at about the same time, very early in the curriculum, may help avoid creating another generation of engineers whose first thoughts about parallelism are that it is scary and always hard.

***Divide and Conquer.*** The idea of "split a problem in two, and perform both in parallel" is arguably both easier to learn and more important to understand than the sequential version that is taught early and often in CS courses.

***Algorithm Analysis.*** A presentation of at least Amdahl's Law (for analyze of the extent to which parallel speedups are limited by sequential bottlenecks) is a natural complement to the usual introductory Big-O coverage.

***Loosely Coupled Components.*** It is increasingly common to approach loosely-coupled asynchronous execution as a natural extension of event-based programming (which itself

has become a common topic only since the introduction of GUI programming in CS curricula). There have been some helpful contributions (for example the event-based Scala Actor framework) that help unify ideas of notifications and messaging. However, these have yet to be made boring enough for routine accommodation into courses.

***Aliasing and Independence.*** The small topics of "aliasing", call-by-reference vs value, and so on, in sequential programming blossom into a huge set of concerns in concurrency: data races, process isolation, memory models, ownership types, etc. We have yet to see an attempt to distill the main ideas into a set of essentials that all students should master. At a minimum though, students should appreciate the tradeoffs that may be leading to a greater prevalence of functional programming styles in concurrent than sequential programming.

***Coordination.*** Synchronization abstractions, including locks, monitors, transactions, blocking channels, barriers, and many others have long been a primary focus of concurrent programming, but one that has historically been subject to bizarre factionalization in which proponents of one particular construct are unable to even communicate with those advocating others. We still await approaches to teaching about coordination algorithms and protocols that provide a concise basis for understanding them without becoming locked into any one of these forms.

***Scalable Data Structures.*** The need access and manage large numbers of data elements inspired the discovery of data structures such as B-Trees, which are now common topics in intermediate and advanced courses in data structures and algorithms. Similar, the need to access and manage data across large numbers of threads has led to the discovery of data structures and algorithms that minimize synchronization, such as non-blocking queues. One can only expect that these become equally routine topics in such courses.

***Numerical Computing.*** Algorithms in support of scientific analysis, graphics, and modeling tend to have enough structure to readily support parallel decomposition, and further, to readily exploit special-purpose hardware such as GPUs. However, there does not yet seem to be a consensus about which of the underlying concepts are central enough for most or all students to learn, as opposed to the realm of specialized courses in these and related areas.

***Systems.*** Architectures, operating systems, virtual machines, compilers, tools, and runtime systems in support of parallel programming are more complex than those for sequential programming. As an unfortunate byproduct, higher-level programmers are increasingly oblivious to even the basic theory of operation of the substrates underlying everyday computing. They are thus unable to cope with performance anomalies or unexpected behaviors. Good solutions probably await good ideas on how to defragment "systems" courses to arrive at offerings that convey essential ideas.