An undergraduate curriculum infused with parallelism

Vijay S. Pai and Samuel P. Midkiff, School of ECE, Purdue University
{vpai,smidkiff}@purdue.edu

During the 1980s, with the rise of the mini-supercomputer, it was a given that parallelism would soon be ubiquitous. When the "killer micro" resulted in the death of most of the mini-supercomputer companies, the arrival of ubiquitous parallelism became less certain, a consensus that existed only until dual processor workstations became popular in the late nineties. For economic reason, however, these machines never became widespread.

Ubiquitous parallelism did finally arrive around the middle of last decade. After repeated predictions that ubiquitous parallelism would soon be upon us, large parts the software industry and the higher education system were unprepared for its actually arrival. The software industry was forced to confront a work force trained in certain languages, and dependent on increasingly faster processors to support increasingly complex features – features which drove the adoption of new, and sometimes even better, versions of software. Computer Science and Engineering departments were forced to confront a curriculum that stressed high-level languages, abstraction and clean code, with little attention paid to architecture and low-level implementation details.

The current focus on parallelism in undergraduate education is driven entirely by architectural changes. The type of parallelism being discussed – typically thread based parallelism relying on a shared address space; and the abstractions used to exploit and control that parallelism – Pthreads, language specific thread models, transaction and locks, are driven entirely by these architectural changes. Moreover, the algorithms taught in, e.g. a data structure course emphasizing parallelism, primarily target shared memory architectures, in part because of these architectural changes. We note that despite radical changes in the underlying architecture, the economic importance of harnessing those changes for general purpose, commodity software, makes a focus on abstraction essential for the software industry to continue to develop and maintain complex systems. The additional complexity caused by having these software systems utilize parallelism for higher performance will only add to the need for abstraction. Fortunately, parallelism offers rich opportunities to show the utility and necessity of abstraction.

It is our belief that an understanding parallelism, and concepts related to parallelism (independence, ordering, atomicity, and so forth) are fundamental to computation. Because of that, it is essential to students be introduced early to these concepts. However, because these concepts are so fundamental, it is no more desirable to teach them in a single "parallelism" course than it would be to teach concepts of abstraction in only a single programming course. Therefore, the teaching of parallelism *must* begin early, and *must* continue throughout the undergraduate curriculum (and the graduate curriculum, for those students that pursue a post-graduate degree.) A practical and severe challenge is that the because software will only get more complex, material currently taught about good software practices must continue to be taught.

At Purdue we are tackling these challenges with a comprehensive evaluation and reform. For the rest of this position paper we will describe our efforts in this area.

Teaching parallelism early. We believe that it is essential to introduce parallelism in the first year because students initially exposed to conventional sequential models of problem-solving will have difficulty adjusting to concurrency since some of their previous assumptions will have to be "unlearned." Additionally, we only have four years to convert high school seniors into functioning computer engineers and scientists and thus must budget that time carefully.

We employ the bottom up model pioneered by Patt and Patel and described in a widely adopted textbook [1]. The course starts with the bare basics of bits and Boolean logic, progresses to numerical representations, moves on to transistor switches and digital logic, builds up instruction-set processing and the computer's data path, discusses I/O, and then introduces structured programming in assembly language before moving on to high-level programming in C.

Teaching, at a high level the hardware components of computing is valuable components of computing have high levels of concurrency as various resources operate in parallel. For example, the ALU can be used to calculate the next program counter value at the same time that the register file is being read to determine the source registers of the current instruction. Consequently, we increase the emphasis on the computer data path and how various elements operate at the same time. This naturally extends to a multicore computer by having multiple computational state machines operating simultaneously. Another exercise we do is to have the student consider asynchronous I/O devices, and then asynchronous devices with their own processor. Control of the low-level hardware with a high level language introduces students to the value of abstraction.

Sophomore courses. At the sophomore level we teach *Advanced C Programming* and *Introduction to Digital System Design*. The digital systems course low-level parallelism in adders and synchronous and asynchronous design. In the programming course, the students become acquainted with different concepts of parallelism, including sockets, `fork` system calls, and in advanced sections data-sharing, Amdahl's Law, message queues and concurrency control. By linking the low-level hardware and software concepts to higher levels abstractions (instructions, library calls and data structures) the principles of information hiding and abstraction are reinforced.

Junior courses. In the junior year *Software Engineering Tools* (a scripting languages course)*Data Structures* and *Microprocessor System Design and Interfacing* are taken. The software courses do, or will, provide some exposure to multi-threading in Python and parallel data structures algorithms. The microprocessors systems course introduces students to hardware and software concurrency in the form of interrupts.

Enhanced senior courses. In the senior year two or more of *Introduction to Digital Computer Design and Prototyping*, *Software Engineering*, *Object-Oriented Programming in C++ and Java* and *Introduction to Compilers and Optimizing Compilers*. The digital computer design course is an introductory architecture course that requires the students to do a VHDL design of a multicore processor and learn about coherency and synchronization issues. All of the software courses discuss parallelism – the first two in terms of parallel and concurrent object oriented programs, and the compilers course in the context of an auto-parallelizing compiler and concepts of independence and dependence.

As with all curricular issues, constant monitoring and revision is necessary. We are guided in this by means of a *Concurrency Content Inventory* test that we give to students at various times during their studies. Initial feedback indicates that our efforts to make parallelism an fundamental part of the entire undergraduate curriculum are paying off.

References

[1] Yale Patt and Sanjay Patel. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. McGraw Hill, 2001.