Concurrency and Parallelism as a Medium for Computer Science Concepts

Steven Bogaerts Kyle Burke Brian Shelburne Eric Stahlberg

Department of Mathematics and Computer Science
Wittenberg University
Springfield, OH 45501
USA
{sbogaerts, kburke, bshelburne, estahlberg}@wittenberg.edu

Abstract

This paper argues that the integration of concurrency and parallelism topics throughout the computer science curriculum need not require a significant reduction in coverage of more "standard" topics. This is accomplished by recognizing that concurrency and parallelism can be used as a medium for learning about other standard topics, rather than as an additional topic to cover. This paper argues this point and describes ongoing work towards it.

Categories and Subject Descriptors K.3.2 [*Computers and Education*]: Computer and Information Science Education – computer science education, curriculum

Keywords computer science curriculum, concurrency, parallelism

1. Introduction

In considering curricula for concurrency and parallelism, it is useful to make an analogy. Consider object-oriented programming (OOP). At its core, OOP is a different paradigm from imperative programming, the primary paradigm in use decades ago. As OOP was developed, we can imagine posing questions similar to those of this workshop: Should OOP be taught in introductory computer science courses? Should OOP topics be "sprinkled" into existing courses?

Of course there are still many variations in curricula, but in general we can see how these questions have been answered. While there is a place for a high-level OOP course, object-oriented concepts are by no means relegated only to such a course. CS1 typically includes the use of objects and basic class construction. A data structures course often includes the creation of abstract data types with classes. Graphics courses can make use of abstraction through classes as well as through procedures. The inclusion of these OO topics has necessitated some additional time to learn mechanics, but it is fair to say that many of the topics of these courses are simply being taught through the medium of OO now, rather than solely through the medium of imperative programming. Furthermore, while perhaps some sacrifices have been made, most key concepts of imperative programming have not been sacrificed to achieve this OOP coverage.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPLASH '10 10/17/2010, Reno, NV.

Copyright © 2010 ACM . . . $10.00

We argue that the same can and will be said for questions of parallelism and concurrency education. There will always be a place for an advanced course in parallelism, but this should not be the only place for such topics. Like OOP, parallelism is too broad-reaching to be limited to a single advanced course. As such, many computer science educators have recommended the "sprinkling" of parallelism throughout the curriculum (e.g., [1], [2]). This paper argues further that the best way to accomplish this while not sacrificing traditional content is to recognize parallelism as a complementary *medium* for learning various computer science topics. It does require some additional background in basic mechanics, but once these basics are covered, parallelism can be used in combination with traditional approaches for learning computer science. The key is that this can be done without significant elimination of other material; rather, other material is simply learned through the medium of concurrency and parallelism.

The computer science faculty at Wittenberg University, in cooperation with colleagues from Clemson University, are working under a three-year National Science Foundation grant[1] with principal investigators Eric Stahlberg and Melissa Smith, to put these ideas into practice in a total redesign of our curriculum. This partnership is a direct response to important challenges facing smaller institutions and departments. Faced with limited resources in both personnel and hardware, it became evident that integration of the fundamentals was essential to eliminate the need to add and staff an elective course. Furthermore, as we argue in this paper, we believe that much traditional course content can be covered through the medium of parallelism and concurrency.

Thus a key component of our work in this grant has gone towards a redesign of content in existing courses to make use of this medium. This redesign is under development for courses across the computer science curriculum, as well as for applications courses in bioinformatics, computational models and methods, and computational chemistry. To illustrate, the remainder of this position paper describes our work in CS1 and programming languages courses, plus some additional thoughts on computer organization courses.

2. Enhancements in CS1

Similar to OOP as discussed above, some background material is required before students can use parallelism as a medium for learning standard CS1 topics. In this section we describe this background material, and then discuss how parallel programming is used as a medium for learning other CS1 topics.

[1] National Science Foundation grant CCF-0915805, SHF:Small:RUI:Collaborative Research: Accelerators to Applications – Supercharging the Undergraduate Computer Science Curriculum

2.1 Background Material

The background material comes in two parts: 1) a high-level overview of parallel computing concepts and technologies, and 2) an introduction to basic required syntax.

The high-level overview includes discussions in:

- How non-technical tasks are often naturally done in parallel.
- Various physical class activities in which the students play the part of cores, messages, etc, to vividly illustrate basic concepts. Some of these activities are based on work in [3].
- The terminology of concurrent and parallel computing.

Depending on constraints and preferences, this material can be covered in one to four hours of in-class time.

Since our CS1 course is taught in Python, the introduction to syntax portion of the background material covers basic use of the Python `multiprocessing` package, which enables the utilization of multiple cores. This package is quite simple syntactically. Within one to two hours of lecture time, students should understand the basic syntax of process creation, communication, and synchronization.

2.2 Parallelism as a Medium for CS1 Material

Once this background material is covered, much remaining instruction in parallel computation can occur in CS1 with very little additional cost, and thus without sacrificing any additional standard CS1 content.

For example, suppose students are learning how to use multiple functions with arguments, to split a larger task into logical chunks. Suppose they are practicing this on a program that computes the roots of a binomial using the quadratic formula. We can imagine breaking this up into a `main`, `computeDiscriminant`, and `computeQuad`. Then the students can be asked what should become a common question: what parts of the computation can be done in parallel? There are many possibilities, but one simple option is to compute simultaneously the ”plus” and the ”minus” part of the numerator in the quadratic formula. This requires modifying `computeQuad` to take a sign argument. `main` should then simply spawn two processes: one calling `computeQuad` with +1, the other with -1.

Thus in this example we see that students have received good practice in splitting a task up into functions and passing needed arguments. The only difference here from a typical CS1 lesson is that students are getting practice in this in both sequential and parallel programming. The additional practice in parallel programming has been obtained without sacrificing any coverage of functions, and without significantly adding to required course time.

To briefly consider another example, a common CS1 exercise is to find the maximum key in a list. Again, students should be asked: how could this be done in parallel? A simple answer is to split the list into `n` segments assigned to `n` processors. Each process finds the largest key in its segment, and a master process finds the ”largest of the largest”, as it were. This simple approach gives students good practice in `if` statements and looping through a list, all in the context of parallel programming. Again, the key is that this practice in parallel programming is obtained in a manner wholly integrated in the ordinary CS1 curriculum, while neither sacrificing the original content nor adding required course time.

3. Enhancements in Programming Languages

A significant component of many programming languages courses is the exploration of various programming paradigms and how languages are designed to facilitate particular tasks within a paradigm. Thus it is a natural fit to integrate high-performance computing (HPC) languages into the course. In studying HPC languages, students can see how new syntax supports multi-threading, alleviates common concurrency issues, and simplifies vital parallel patterns. Language features such as these mirror core differences to imperative languages studied via the logic and functional paradigms.

In spring 2010 we integrated discussion of the HPC language Chapel [4] into our programming languages course. Much of the syntax is similar to imperative languages, so students were able to jump right into the parallel tools. They enjoyed working with the added language features, such as timers, variable access controls, and the built-in reduce and scan operations.

The assimilation of HPC languages into the programming languages course enabled attention to parallel programming while still remaining focused on a core mission of the course: to learn about various programming paradigms and language design to support particular functions. The new HPC languages, such as Chapel, provide an excellent medium for exploration of these concepts.

4. Enhancements in Computer Organization

It should be noted that computer organization courses implicitly include parallelism. Parallelism is seen at the digital logic level in combinatorial circuits, at the instructional level in pipelines and super-scalar architectures, and in the way I/O is done using interrupts and DMA. Thus integrating parallelism into a computer organization course is simply a matter of making the inherent parallelism more explicit. Additional opportunities for demonstrating parallelism can be obtained by introducing ”parallel” hardware design languages such as VHDL (VHSIC (Very-High-Speed Integrated Circuit) Hardware Description Language) [5]. For example, VHDL can be used to describe complex parallel circuits like a carry look-ahead adder, with its parallel propagation of ”carrys”. Comparisons can also be made to the more serial approach of the ripple carry adder. Furthermore, earlier exposure to hardware description langauges like VHDL and Verilog [6] can pave the way for later courses in architecture and hardware design.

5. Conclusion

In this position paper we have argued that, while some new material is required, students can become very proficient in concurrency and parallelism within the context of the standard CS curriculum. With careful integration of these topics, this can be accomplished without a significant reduction in material covered in other areas. Rather, concurrency and parallelism can serve as a medium in which many standard topics are discussed.

References

[1] Nevison, C. H. 1995. Parallel Computing in the Undergraduate Curriculum. In *Computer*, 28(12): p. 51-56. IEEE Computer Society.

[2] Meredith, M. J. 1992. Introducing Parallel Computing into the Undergraduate Computer Science Curriculum: A Progress Report. In *ACM SIGCSE Bulletin*, 24(1): p. 187-191. New York, NY, USA: ACM Press.

[3] Maxim, B. D.; Bachelis, G.; James, D.; and Stout, Q. 1990. Introducing Parallel Algorithms in Undergraduate Computer Science Courses. In *Proceedings of the Twenty-First SIGCSE Technical Symposium on Computer Science Education*, p. 255. New York, NY, USA: ACM Press.

[4] Chamberlain, B. L.; Callahan, D.; Zima, H. P. 2007. Parallel Programmability and the Chapel Language. In *International Journal of High Performance Computing Applications*, 21(3): p. 291-312.

[5] Perry, D. L. 1993. VHDL (2nd Ed.). New York, NY, USA: McGraw-Hill, Inc.

[6] Thomas, D. E.; Moorby, P. R. 1998. The Verilog Hardware Description Language (4th Ed.). Norwell, MA, USA: Kluwer Academic Publishers.