Addressing Roadblocks in Parallel Programming

Steven Bogaerts, Brian Howard, Maria Schwartzman, Scott Thede, Gloria Townsend
Department of Computer Science
DePauw University
Greencastle, IN, U.S.A.
{stevenbogaerts, bhoward, mariaschwartzman, sthede, gct}@depauw.edu

***Abstract*—In this document we propose a poster on our efforts to integrate parallelism into a wide range of computer science courses at DePauw University. We focus on multi-semester experimental results in CS1, including two redesigns with a final reintegration of a significant multithreaded programming component, leading to improvements in student learning and interest outcomes. We also provide brief comments on new integrations of parallelism in a data mining course.**

***Index Terms*—parallelism; CS education; curriculum**

We have new evaluation results for our **CS1** module, which combines illustration by analogy with programming in Java using a CS1Thread class. NSF/TCPP curriculum topics include Architecture (Multi-Core: K), Programming (Shared Memory: C, Distributed Memory: C, Task/Thread Spawning: A, Tasks and Threads: C, Synchronization: C, Concurrency Defects: C, Performance Metrics: K) and Cross Cutting (Why and What is PDC?: C, Non-Determinism: K, Power Consumption: K). Our experiments include three contrasting approaches (A, B, and C) to the programming portion of the module.

Approach A includes Java multithreaded programming, with problem decomposition and resolution of simple race conditions. This approach was difficult, as we observed that students needed to wrestle with *three simultaneous challenges*: (1) common CS1 topics (e.g. object composition), (2) Java-specific issues of multithreading (e.g. Runnable vs. Thread), and (3) the implementation of key parallelism ideas (e.g. race condition resolution). Student ratings of the difficulty of these three topics support the idea that each are a source of challenge. This confluence of topics, along with the limited time, made hands-on exercises difficult in approach A, and so the content was covered rather quickly in a lecture format.

As a result, we attempted approach B, reducing the programming content and focusing in the module on what there was time to do particularly well. Unfortunately, this significant reduction of the programming content corresponded to a measured reduction in *non-programming* post-test scores, from 79% to 59%. This suggests that the programming content played an important role in helping students understand even the non-programming content. Evaluation results also suggest a reduction in student desire to learn more about parallelism. Under approach B, on a 5-point scale, there was an average reduction of 0.54 points in students' self-reported desire to learn more, while the same measure held steady under approach A. With no other changes in the parallelism module, these results suggest that the programming component was an integral part of maintaining students' interest and helping them learn even the non-programming material.

In light of these results, the module was revised again for approach C. We returned to significant programming, but redesigned to address the three simultaneous challenges described above. Students were first given a programming assignment on Java multithreading *ideas* without actually using threads. This may sound contradictory, but actually many ideas needed in Java multithreaded programming can be explored in a purely sequential context first (e.g., array decomposition). With such practice in a sequential context, students face fewer simultaneous challenges when moving to the multithreaded context. Follow-up parallelism assignments are analogous to the initial sequential one, thus reducing the number of challenges to be faced at once.

In approach C, with a renewed programming focus, the same *non-programming* pre- and post-tests were given as in previous semesters. The average post-test score was 74%, compared to the 59% for approach B and 79% for approach A. However, approach C was also the first to show an increase in students' desire to learn more about parallelism. This suggests that the step-by-step approach of C helped students see the interesting challenges to be explored while not overwhelming them with many details simultaneously.

Another new effort in our NSF/TCPP curriculum adoption is in the **data mining** course. This is an upper-level undergraduate major course, considering a wide range of data pre-processing and machine learning techniques. The course uses entirely real-world datasets. Thus, parallelism is an important component of the work, as sequential computational cost quickly becomes significant. The course uses Python with common modules including Pandas, SciPy, NumPy, and Scikit-Learn. Many operations in these modules support parallelism through parameters like `n_jobs`, in which the programmer can specify the number of processes to use in executing an operation. This only makes sense for significant operations, of course, such as k-fold cross validation and large maps, filters, and reductions. For other computations, for which a single operation with an `n_jobs` parameter is not applicable, finer-grained control via the `multiprocessing` module and a pool of threads can be applied.

Space limitations preclude discussion of **other courses** here, but the poster will also include summary information on our efforts in an interdisciplinary team-taught course, along with courses in data structures, programming languages, computer systems, foundations of computation, and parallel computing.