PDC Modules for Every Level: A Comprehensive Model for Incorporating PDC Topics into the Existing Undergraduate Curriculum

EduPar-11 Early Adoption Report, 2013 Update, 15 February 2013

Joseph P. Kaylor, Konstantin Läufer,[1] Chandra N. Sekharan, and George K. Thiruvathukal
Department of Computer Science, Loyola University Chicago, laufer@cs.luc.edu

Overview

In this brief update, we inform the TCPP Curriculum Committee of our continued efforts as early adopters. During 2011, we implemented seven three-week PDC course modules, each of which we incorporated into a different existing course, ranging from CS2 to advanced electives. ***During 2012, we have focused on pushing PDC topics further down into CS1 and CS2, fleshing out PDC coverage in our intermediate object-oriented development course (CS 313), and stepping up evaluation.***

Background: Efforts Prior to EduPar-11

These are our pre-EduPar-11 efforts to incorporate PDC topics into the undergraduate curriculum:

- We offered a single course in concurrent programming in a functional/object-oriented environment aimed at sophomores (preferably in their fourth semester) from 1997 until 2007.[2]
- We then broke up this body of knowledge into units taught regularly as part of our range of existing advanced/elective courses; this is consistent with the stated TCPP rationale.

EduPar-11 Outcome (I): Required Core Modules

In an effort to expose all undergraduate majors to PDC topics regularly, consistently, and early on, we have created three three-week core PDC modules (20% of our 15-week semester) included in required foundation courses taken in the second year and mostly at the “K” and “C” level:

- *Introduction (CS2 course):* parallel control statements; shared memory language extensions and libraries; tasks, threads, and synchronization; and performance considerations.
- *Programming (CS 313):* Paradigms (threads) and related semantics and correctness issues including synchronization and defects. (See below for recent updates to this module.)
- *Architecture (CS 264: Intro Sys Arch):* The various dimensions of the architectural design space, memory hierarchy, data representation, and performance metrics.

EduPar-11 Outcome (II): Advanced/Elective Modules

Several advanced modules in programming and distributed computing, typically offered every three semesters, have been developed and incorporated into the courses listed. Topics are covered mostly at the “C” and “A” levels.

- *Advanced Programming:* programming languages course (CS 372, in Haskell or F#) and advanced object-oriented programming course (CS 373, in Scala)
- *Distributed Foundations:* distributed systems course (CS 339)
- *Distributed Programming and Applications:* web services course (CS 342)
- *Additional PDC topics (two 3-week modules):* operating systems course (CS 374)

[1]Department chair and corresponding author
[2]C. Colby, R. Jagadeesan, K. Läufer, and C. Sekharan. Interaction, Concurrency, and OOP in the Curriculum: a Sophomore Course. *OOPSLA 1998 Educators' Symposium and Poster.*

EduPar-11 Outcome (III): 2013 Update • Focus on the Intro Sequence

Recent changes in the environment of Loyola University Chicago's Department of Computer Science include a clear differentiation among our four undergraduate majors, growing interest in computing among science majors, and an increased demand for graduates with mobile and cloud skills. In our continued effort to push parallel and distributed computing topics further down into the introductory sequence, we are focusing on these three existing courses:

CS1: In response to a request from the physics department, we started to offer a CS1 section aimed at majors in physics and other hard sciences this spring 2013 semester. This section includes some material on numerical methods at the K and C levels, and about 9 class hours will be dedicated to sequential and parallel versions of these algorithms and the possible resulting speedup, using data parallelism in C#. For example, we can use threads for speeding up trapezoidal rule integration.

```
for (i = 0; i < numThreads; i++) { // create and start new child threads
  its[i]=new IntegTrap1Region(start, end, granularity, fn);
  childThreads[i] = new Thread(new ThreadStart(its[i].run));
  childThreads[i].Start();
  // set the range for the next thread
  start = end;
  end = a + ((i + 2.0d) / numThreads * range);
}
for (i = 0; i < numThreads; i++) {
  childThreads[i].Join(); // wait for child threads to finish
  totalArea += its[i].getArea();
}
```

CS2: We have emphasized PDC topics in CS2 as of fall 2011. The course now includes an 18-hour double module on task parallelism, speedup, and load balancing in algorithms involving arbitrary precision arithmetics. We present these topics at the C and A levels in the form of various examples. For example, we can compute Fibonacci numbers based on repeated squaring of 2-by-2 matrices of BigIntegers in Java, experiment with the speedup resulting from executing lines (3) and (4) in separate threads, and explore load balancing between these unequal tasks.

```
public BigInteger[] matMultFib(final BigInteger[] fibK) {
  final BigInteger[] matFib2K =new BigInteger[2];
  matF[0] = fibK[0].multiply(fibK[0]).add(fibK[1].multiply(fibK[1])); // (3)
  matF[1] = fibK[1].multiply(fibK[0].shiftLeft(1).add(fibK[1]));      // (4)
  return matFib2K;
}
```

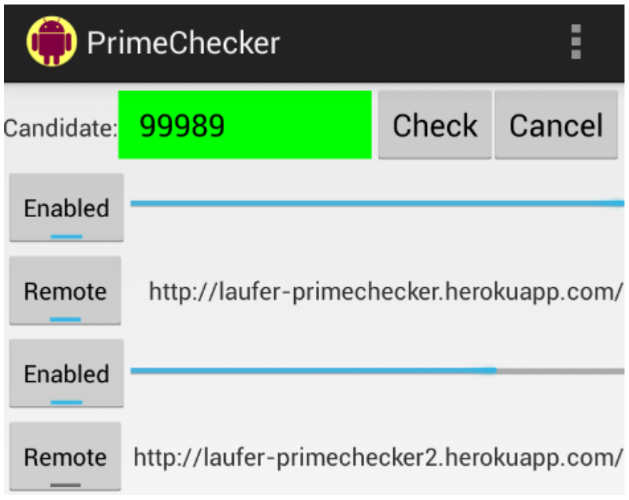
Intermediate OO Development (CS 313): We have emphasized PDC topics in this intermediate, object-oriented software design course since fall 2011. As of fall 2012, we switched the programming projects from C# to Java with Android. The latter provides a highly effective context for studying concurrency and distributed computing topics at the C and A levels. Our 9-hour PDC module covers external and internal events, background threads, offloading computation from the mobile device to the cloud,[3] and observing the resulting throughput-latency tradeoff. The following example, based on a brute-force prime number checker,

[3] Jason H. Christensen. 2009. Using RESTful web-services and cloud computing to create next generation mobile applications. In *Proc. OOPSLA '09*. ACM, New York, NY, USA, 627-634. DOI=10.1145/1639950.1639958.

illustrates these topics, as well as practical considerations such as task cancellation and progress reporting; for sufficiently large primes, the remote task returns almost instantaneously, while the local one is still churning (see screenshot below).

```
// local task
for (long k = 2; k <= half; k += 1) {
  if (isCancelled()) break;
  publishProgress((int) ((k / dHalf) * 100));
  if (i % k == 0) return false;
}
return true;

// task to invoke essentially the same code on the remote side
final HttpResponse response = client.execute(request);
final int status = response.getStatusLine().getStatusCode();
```



http://laufer-primechecker.herokuapp.com/
http://laufer-primechecker2.herokuapp.com/

Pervasive Computing Capstone: Beyond our three regular courses, we have rolled out a research capstone course in which students further develop ideas from CS 313 and subsequent courses.

Evaluation

Evaluation of our PDC course modules will include at least one quiz or exam where the students' understanding of the covered topics is measured using a pre- and post-test design. During 2012, we have made progress toward unified evaluation instruments for certain modules. As a representative example, we have included the instrument for the advanced programming module in the appendix.

Future Plans

- *Algorithms:* Our data structures and algorithms course (CS 363) is offered every fall. We are developing a suitable module that includes the following topics: models of computation and complexity, basic algorithmic paradigms, and specific problems and their algorithmic solutions.
- *Evaluation:* Once our course modules have stabilized, we will need to measure their effectiveness longitudinally over a three- to five-year period. We also intend to refine our current evaluation approach by working with Loyola's Center for Science & Math Education, as well as the TCPP and fellow early adopters.

- *Dissemination:* We consider holding workshops for subsequent adopters in the Midwest.

Appendix: Evaluation Instrument

We have used following evaluation instrument in conjunction with the concurrency module in the programming languages (CS 372) and advanced object-oriented programming (CS 373) courses.

- Suppose we have a soccer stadium with two entrance doors and need to keep track of the number of spectators currently inside. We use a shared mutable variable count for this purpose. Whenever a spectator enters, the following steps are executed, with the intent of incrementing the shared count:
  ```
  let inc() = {
      let local = !count
      count := local + 1
  }
  ```
  Suppose the stadium is empty, count is zero, and two spectators enter at the same time through different doors. What is the conceptually correct value of count once the two spectators are inside?
- Under the same scenario, and given our implementation of inc, what are the possible resulting values for count? Which one(s) are conceptually correct?
- Still under the same scenario, one possible ordering of the four steps corresponding to the two invocations of inc is
  ```
  let local1 = !count // f1
  count := local1 + 1 // s1
  let local2 = !count // f2
  count := local2 + 1 // s2
  ```
  where local1 and local2 are distinct local variables associated with the two doors, respectively. Is the following ordering possible?
  ```
  let local1 = !count // f1
  count := local1 + 1 // s1
  count := local2 + 1 // s2
  let local2 = !count // f2
  ```
- List all other possible orderings of these steps, using the abbreviations f i for fetch i and s i for set i as shown in the comments. (That is, you would list the ordering from the previous subproblem as "f1 s1 s2 f2".)
- What is the root cause of the problem observed here? (check one)
  - choice of a functional programming language
  - use of a shared mutable variable
  - use of a simple type instead of a data structure
  - use of local variables
- What kind of mechanism can you use to ensure that only correct orderings of these steps occur? (check all that apply)
  - encapsulate the shared count inside a thread-safe data structure
  - use an explicit locking mechanism in inc to enforce mutually exclusive access to the shared count
  - use an imperative programming language
  - use message passing instead of shared memory
- Suppose we have two philosophers. The first one, Kant, behaves like so:
  a. *think for 10 minutes*

  b. *wait for any available fork and grab it when available*
  c. *think for 2 minutes*
  d. *wait for any available fork and grab it when available*
  e. *eat for 5 minutes*
  f. *release both forks*

  The second one, Heidegger, behaves like so:

  a. *think for 11 minutes*
  b. *wait for any available fork and grab it when available*
  c. *think for 2 minutes*
  d. *wait for any available fork and grab it when available*
  e. *eat for 5 minutes*
  f. *release both forks*

  Suppose Kant and Heidegger sit at the same table with two forks available and start their respective behaviors at the same time. Give an event trace, that is, a precise description of what happens in the form of observable events such as:

  a. *At minute 4, Kant takes fork 1.*
  b. *At min*ute 7, Heidegger releases fork 2.
  c. ...

- What type of undesirable situation does this scenario illustrate? (check one)
  - lack of thread safety
  - run-time type error
  - memory leak
  - deadlock
- What are possible ways of avoiding this kind of undesirable situation? (check all that apply)
  - use an explicit locking mechanism to enforce mutually exclusive access to each fork
  - provide at least one more fork
  - treat both forks as a single resource bundle that must be acquired together at the same time
  - provide a fork and a knife instead of two forks and rewrite the behaviors such that each philosopher must acquire the fork first and then the knife
- Suppose there are *three* forks instead of two. Suppose Kant and Heidegger sit at the same table with the three forks available and start their respective behaviors at the same time. Give an event trace, that is, a precise description of what happens in the form of observable events such as:

  a. *At minute 4, Kant takes fork 1.*
  b. *At minute 7, Heidegger releases fork 2.*
  c. *...*