5-20-2013

Spring11: PDC in CS1/2 and a mobile/cloud intermediate mobile/cloud intermediate software design course

Joseph P. Kaylor

Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

Chandra N. Sekharan
Loyola University Chicago

George K. Thiruvathukal
Loyola University Chicago

Recommended Citation

J. Kaylor, K. Läufer, C. N. Sekharan, and G. K. Thiruvathukal. Spring11: PDC in CS1/2 and a mobile/cloud intermediate mobile/cloud intermediate software design course. In Proc. 3rd NSF/IEEE-CS TCPP Workshop on Parallel and Distributed Computing Education (EduPar), Boston, Massachusetts, USA, May 2013.

Spring-11: Introducing PDC topics into CS1/2 and a Mobile- and Cloud-Based Intermediate Software Design Course

EduPar-11 Early Adoption Report, 2013 Update, 15 March 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. Prior to this update, during spring 2011, we implemented three three-week PDC course modules (20% of our 15-week semester) targeting three required courses usually taken in the second year. Then, during AY 2011-12, we implemented four three-week advanced PDC course modules in programming and distributed computing targeting electives typically offered every three semesters.[2]

Currently, during AY 2012-13, we are moving 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.

Details of Our Current Focus on the Three-Course 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 (BS in Computer Science, Software Engineering, Information Technology, and Communication Networks and Security), 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;
```

[1] Department chair and corresponding author
[2] Details are found in our papers in the EduPar 2011 and 2012 workshop proceedings, respectively.

```
  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 starting in fall 2011. The course now includes a 9-hour PDC 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 Object-Oriented Development (CS 313): We have emphasized PDC topics in this intermediate object-oriented software design and development course since fall 2011. As of fall 2012, we switched the programming projects from C# back 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 double 18-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, 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 Figure 1 below).

```
// local task: i is the number to check
final long half = i / 2;
final double dHalf = half;
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();
```

[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.

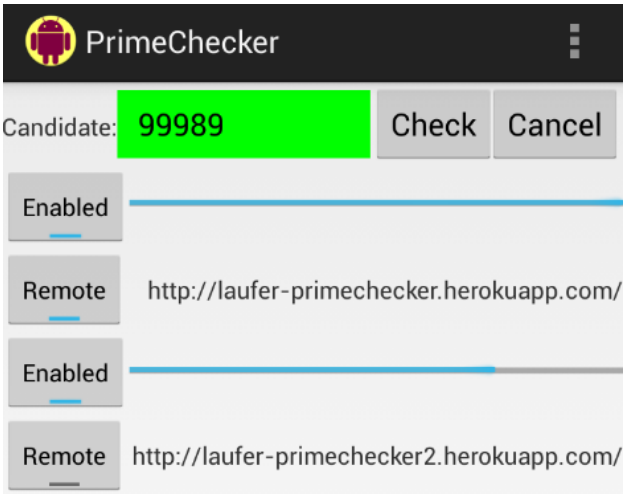

Figure 1: *Partial screenshot of completed remote prime checking task and ongoing local one*

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 proficiency assessment instruments for certain modules. As representative examples, we have included the proficiency assessment instrument in concurrency for the advanced programming module and the course effectiveness evaluation instrument for CS 313 in the appendix.

We have conducted the concurrency assessment twice so far with the following results (normalized to 100 points), suggesting that the material is challenging but a decent command can be imparted through the module we have developed.

Statistic	**Count**	**Min**	**Max**	**Range**	**Avg**	**Med**	**Stdev**	**Variance**
Fall 2010	14	60	100	40	76	71	13	167
Spring 2012	9	63	87	24	80	83	7	56

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 A: Proficiency Assessment Instrument for Concurrency Topics

We have used following assessment instrument in conjunction with the concurrency module in the programming languages (CS 372) and advanced object-oriented programming (CS 373) courses. We welcome feedback on this instrument during or after the workshop.

- 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. *…*

Appendix B: Course Effectiveness Evaluation Instrument for CS 313

We have prepared the following evaluation instrument to measure the effectiveness of the new format of our Intermediate Object-Oriented Development (CS 313) course, based on Riley.[4] The rating questions use a suitable five-point Likert scale. We are now evaluating the fall 2012 section and hope to have a sufficient number of responses ahead of the May 2013 EduPar workshop. We will also evaluate the ongoing spring 2013 section using this instrument. We welcome feedback on this instrument during or after the workshop.

- Your overall GPA at Loyola
- Your approximate computer science GPA at Loyola
- Your class standing
- Rate your Java expertise BEFORE taking this course
- Rate your Java expertise AFTER taking this course
- Rate your Android expertise BEFORE taking this course
- Rate your Android expertise AFTER taking this course
- Rate your agile development (testing, refactoring, pair programming) expertise BEFORE taking this course
- Rate your agile development (testing, refactoring, pair programming) expertise AFTER taking this course
- Rate your software architecture and design expertise BEFORE taking this course
- Rate your software architecture and design expertise AFTER taking this course
- Rate your event-based programming expertise BEFORE taking this course
- Rate your event-based programming expertise AFTER taking this course
- Rate your thread-based concurrency expertise BEFORE taking this course
- Rate your thread-based concurrency expertise AFTER taking this course
- Rate your cloud computing expertise BEFORE taking this course
- Rate your cloud computing expertise AFTER taking this course
- Rate your feeling of preparedness for the job market BEFORE taking this course
- Rate your feeling of preparedness for the job market AFTER taking this course
- Rate the effectiveness of the projects for learning agile development
- Rate the effectiveness of Android for learning software architecture and design
- Rate the effectiveness of Android for learning event-based programming
- Rate the effectiveness of Android for learning thread-based concurrency
- Rate the effectiveness of Android for learning about cloud-based services
- Please provide any suggestions or other comments here

[4] Derek Riley. 2012. Using mobile phone programming to teach Java and advanced programming to computer scientists. In Proc. 43rd ACM Tech. Symp. Comp. Sci. Education (SIGCSE '12). ACM, New York, NY, USA, 541-546. DOI=10.1145/2157136.2157292.