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. During AY 2012-13, we moved PDC topics further down into CS1 and CS2, fleshed out PDC coverage in our intermediate object-oriented development course (CS 313), and stepped up evaluation.

Most recently, during AY 2013-14, we have 1) prepared for enhanced PDC coverage in a mathematically focused version of CS1 (CS 215) to be offered in this form during AY 2014-15 and 2) expanded PDC coverage in our programming languages course (CS 372). These efforts are synergistic with our involvement in the SIGCSE 2013 and 2014 workshops on using the Scala language across the curriculum and in CS1/2, respectively. In particular, the use of Scala in CS1/2 provides opportunities to introduce PDC topics early, in accessible yet rigorous ways.

PDC Topics in the Mathematically Focused Version of CS1 (CS 215)

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), strong and sustained enrollment growth across these majors, growing interest in computing among science majors, and an increased demand for graduates with skills in mobile and cloud computing.

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 in the spring 2013 semester. This section included some material on numerical methods at the K and C levels, and about 9 class hours are dedicated to sequential and parallel versions of these algorithms and the possible

---

1 Department chair and corresponding author
2 Details are found in our entries in the EduPar 2011-2013 workshop proceedings, respectively.
resulting speedup, using data parallelism in C#. For example, we used threads for speeding up trapezoidal rule integration.

```csharp
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();
}
```

Based on our experience of working with the physics and math/stat departments, we are now expanding beyond the explicit threading example shown in the preceding discussion. Our involvement as co-authors of two SIGCSE Scala workshops so far is part of a general rethinking about how we can teach concurrent and parallel principles more effectively, especially if we limit or altogether eliminate side effects. Toward this end, we are bringing back ideas from our dissertations, which were focused on functional programming and type systems (Läufer) and functional dataflow (Thiruvathukal). These ideas will be brought into our forthcoming offering of CS 215, a joint mathematically and scientifically-focused course with the math/stat department that students can take in place of our main CS1 offering (CS 170).

Contrast the above explicit threading example from C# with this Scala version:

```scala
def integrateSequential(a: Double, b: Double, rectangles: Int, f: Fx): Double = {
  val interval = (b - a) / rectangles
  val fxValues = (1 until rectangles).view map { n => f(a + n * interval) }
  interval * (f(a) / 2 + f(b) / 2 + fxValues.sum)
}
```

The most important thing to note here is that this version is purely functional. That is, it has no side effects. Note that `val` is used to initialize `interval` and `fxValues`, after which no further mutation is possible. In addition, it is noteworthy that the code actually matches the mathematics for how trapezoidal integration is computed. While this might not seem like a big deal, we know that it is a lot easier to parallelize a code when we can understand (comprehend) the code and see whether natural opportunities for expressing parallelism exist. In this code, the opportunity is in the iteration over the regions (rectangles). So going from sequential to parallel in this code requires only a simple change to create a parallel collection:

```scala
def integrateParallel(a: Double, b: Double, rectangles: Int, f: Fx): Double = {
  val interval = (b - a) / rectangles
  val fxValues = (1 until rectangles).par.view map { n => f(a + n * interval) }
  interval * (f(a) / 2 + f(b) / 2 + fxValues.sum)
}
```

The `par` is highlighted in bold to show the power at our disposal. If you are going to teach

---

something at the CS1 level, it needs to be clear, concise, and show actual benefits. (Adding par does in fact speed up the code proportional to the number of processors. While not as fast as native C, the tradeoff between comprehension and performance is well worth it at this stage and gives a proper introduction to parallel concepts. We can achieve this with the parallel-for in C# and other languages, but we think the clarity of this code is not in doubt.

Lastly, because Scala supports call-by-name style parameter passing and can accept any block of code, there is a highly desirable pedagogical feature that arises. We can provide our students with a method for timing, which can then be used to time any computation (or group of computations) accurately without having to reinvent the wheel (as is often done in many HPC codes):

This method shows how to time a block of code and ensure the result can still be used by a subsequent computation:

```scala
def timeThis[A](s: String)(block: => A): A = {
  val time0 = System.currentTimeMillis
  val b = block
  val time1 = System.currentTimeMillis - time0
  println("Timing " + s + " = " + time1)
  b
}
```

This shows how to time our different integration methods. The student can vary the problem size by using the command-line (perhaps in a script):

```scala
def main(args: Array[String]) {
  try {
    require { 2 <= args.length }
    val rectangles = math.max(args(0).toInt, 1000)
    val n = math.max(args(1).toInt, 1)
    val grainSize = if (args.length == 3) math.min(args(2).toInt, rectangles) 
      else rectangles
    timedRun(rectangles, n, "sequentially", integrateSequential)
    timedRun(rectangles, n, "in parallel", integrateParallel)
    timedRun(rectangles, n, "in parallel with " + grainSize + 
      " rectangles per serial worker", integrateParallelGranular(grainSize))
  } catch {
    case _: NumberFormatException => usage()
    case _: IllegalArgumentException => usage()
  }
}
```

We believe that the Scala-based approach proposed here will allow us to expand the number of hours dedicated to PDC topics in CS1 from 9 to at least 12 and step up the level of
learning to C and, for some topics, A. We expect to provide evaluation results in our next update.

**PDC Topics in a Programming Languages Course (CS 372)**

After a six-year hiatus owing to low enrollments in our department, we resumed our regular programming languages course offering in fall 2010, initially on a three-semester cycle and now on a yearly cycle starting this spring 2014.

The course has our CS 313 course (summarized below) as a prerequisite. Therefore, it does not cover the object-oriented programming paradigm in depth. Instead, it focuses on extensive coverage of the functional programming paradigm and explores relevant PDC topics in this context.

Concretely, the course comprises the following modules:

- Functional programming (6 weeks): higher-order functions (A), algebraic data types (A), Scala OO features (A), recursion patterns (A), the scalaz library for higher-order abstractions (C/A). In this module, students learn how to manipulate immutable linear and nonlinear data using higher-order techniques; this sets the stage for the later discussion of PDC topics.
- Programming language representation and implementation (3 weeks): interpreters (A), parsers (C/A), domain-specific languages (C). In this module, students learn a subset of key compiler topics by implementing an interpreter for a call-by-name lambda calculus.
- Concurrent and parallel programming (4.5 weeks): mutable versus immutable state (C), asynchronous/reactive programming (A), futures and promises (A), threads (C), actors (C/A), other techniques (K/C): software-transactional memory, task-parallel library, functional data structures. In this module, students learn several state-of-the-art concurrent/functional programming techniques and apply them in distributed computing situations such as aggregating content from different web services.
- Other paradigms (1.5 weeks): logic programming, scripting.

The following example gives the reader a sense of the focus on basic, composable algebraic structures as a foundation for studying PDC topics. Specifically, functors are structures with a map operation that can usually be parallelized, depending on actual type of structure. In addition, monoids (semigroups with an identity element) enable parallel reduce. The two together give us map-reduce. These abstractions can be found in Scalaz, a relatively well-known third-party library for Scala.

Through its support for higher-kindred types, similar to that found in Haskell, Scala allows us to define types for these structures and operations on them. In the first three sections of the example, we can see that many common concrete types are naturally instances of these abstractions. Then we define a `twice` method that accepts any monoid value and combines it with itself. Higher-kindred typing enables a degree of abstraction and resulting code reuse that is not possible to the same extent in conventional statically typed languages. Finally, Scalaz has bindings for automatic testing of universally quantified properties, such as the monoid's identity and associativity laws.
In addition, Scala has support for expressing and composing concurrent, asynchronous computations in the form of futures. In the following example, the goal is to add only those numbers in a given sequence that are prime. We create futures to check asynchronously whether each candidate is prime. Because we need all the results for the subsequent steps, we convert the sequence of futures to a future of a sequence, on which we then perform the remaining steps without ever leaving the realm of asynchronicity. Only at the very end do we wait for and print the final result.

```scala
def twice[T: Monoid](x: T) = x |+| x
twice(3) assert === 6
twice(List(1, 2)) assert === List(1, 2, 1, 2)
twice("hello") assert === "hellohello"
```

```
monoid.laws[Int].check
monoid.laws[List[Int]].check
monoid.laws[String].check
```

**Summary of Other Targeted Courses**

**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 multithreading, and explore load balancing between these unequal tasks.

**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, and observing the resulting throughput-latency tradeoff. The poster includes various key examples.

**Pervasive Computing Capstone:** Beyond our regular courses, we started offering a new yearly research capstone course in spring 2013. Here, students further develop ideas from CS 313, CS 372, distributed computing, and other related courses. The emphasis of this course is on developing living distributed/embedded systems that can be installed and showcased in the department and serve as a context for exploring research questions in PDC and other relevant areas. In these projects, we generally try to combine commodity hardware such as Android tablets, Raspberry Pis, and Phidget sensors in novel ways. Current projects include a video wall composed of multiple interconnected Android tablets, an environmental sensing and control system, and a distributed generative music system.

**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 three times 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.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Course</th>
<th>Instrument</th>
<th>Count</th>
<th>Min</th>
<th>Max</th>
<th>Range</th>
<th>Avg</th>
<th>Med</th>
<th>Stdev</th>
<th>Var</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fall 2010</strong></td>
<td>372</td>
<td>1 + 2</td>
<td>14</td>
<td>60</td>
<td>100</td>
<td>40</td>
<td>76</td>
<td>71</td>
<td>13</td>
<td>167</td>
</tr>
<tr>
<td><strong>Spring 2012</strong></td>
<td>372</td>
<td>1 + 2</td>
<td>9</td>
<td>63</td>
<td>87</td>
<td>24</td>
<td>80</td>
<td>83</td>
<td>7</td>
<td>56</td>
</tr>
<tr>
<td><strong>Fall 2013</strong></td>
<td>313</td>
<td>1 only</td>
<td>results to be added before the workshop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Spring 2014</strong></td>
<td>372</td>
<td>1 + 2</td>
<td>results to be added before the workshop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 PDC topics: models of computation and complexity, basic algorithmic paradigms, and specific problems and their algorithmic solutions. This item has been under future plans for several iterations because of instructor changes, and we hope to make progress on it this fall 2014.

- **Evaluation:** We will continue to measure the effectiveness of our PDC modules 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:** Discussions are underway about holding workshops for subsequent adopters in the Midwest in collaboration with other early adopter groups, possibly in the summer of 2015.
Appendix A: Proficiency Assessment Instrument for Concurrency Topics

We have been using the following assessment instrument in conjunction with the concurrency module in the programming languages (CS 372) and advanced object-oriented programming (CS 373) courses since fall 2010. For fall 2013, we split the instrument in two sections, the first of which we used in the intermediate object-oriented design course (CS 313). In spring 2014, we plan to use both sections again in programming languages (CS 372) and possibly add a third section on concurrency in functional languages.

We welcome feedback on this instrument during or after the workshop.

Section 1: race conditions

- 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:

```markdown
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

```markdown
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?

```markdown
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)
  o choice of a functional programming language
  o use of a shared mutable variable
  o use of a simple type instead of a data structure
  o 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

Section 2: deadlock
- 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 minute 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: New Course Effectiveness Evaluation Instrument for CS 313

For fall 2013, we designed the following evaluation instrument to measure the effectiveness of the new format of our Intermediate Object-Oriented Development (CS 313) course, based on earlier work by Riley. Our instrument is incorporated in a more general form used in the IDEA Center’s standardized learning assessment process the university adopted effective fall 2013. We will also evaluate our ongoing spring 2014 sections using similar instruments.

We welcome feedback on this instrument during or after the workshop.

Evaluation scales and questions
Describe the amount of progress you made on the following course-specific learning objectives by using the following scale:

1 = No apparent progress
2 = Slight progress
3 = Moderate progress
4 = Substantial progress
5 = Exceptional progress

48. Acquiring further experience in Java programming
49. Gaining experience in Android mobile application development
50. Gaining experience in agile development (testing, refactoring, pair programming)
51. Acquiring an appreciation of software architecture and design
52. Learning how to design and implement event-based applications
53. Understanding the basics of thread-based concurrency
54. Understanding the role of cloud-based computing in mobile application development
55. Preparing for the job market in computer science/software development

Describe the effectiveness of the following learning mechanisms by using the following scale:

1 = Strongly disagree
2 = Disagree
3 = Neutral
4 = Agree
5 = Strongly agree

56. The programming projects provided an effective context for learning agile development
57. Android provided an effective context for learning event-based programming
58. Android provided an effective context for understanding cloud-based programming
59. Android provided an effective context for appreciating software architecture and design

Evaluation results
The two images included below show the relevant sections of the evaluation results for

---

undergraduate and graduate students, respectively, in a single combined section.

The discrepancy in ratings between the two populations suggests that these topics continue to be challenging for undergraduates. Not all of our undergraduates, however, are consistently exposed to PDC topics in their prerequisite CS1 and CS2 sections. We hope to address this problem by increasing the consistency of the early coverage of PDC topics.