Lecture-less Java-Threads Training in an Hour?

Prasun Dewan
Department of Computer Science
University of North Carolina
Chapel Hill, USA
dewan@cs.unc.edu

***Abstract*—Introducing concurrent execution, forking, joining, synchronization, and load balancing of Java threads to trainees allows them to (a) create arbitrary concurrent algorithms, and (b) be exposed to the underpinnings of concurrency concepts. However, it requires the sacrifice of some existing concepts in the course in which such training is added. To keep this sacrifice low, we ambitiously explored if such concepts can be effectively introduced and tested in a single class period, which is approximately an hour, without a live lecture. Students were asked to learn the concurrency concepts by reading, running, fixing, and testing an existing concurrent program, and taking a quiz. They had varying knowledge of concurrency and Java threads but had not implemented concurrent Java programs. Both in-person and remote help were offered. They were allowed to finish their work after class, within a week. The vast majority of them who started on time finished the coding correctly and gave satisfactory quiz answers in ninety minutes. This experience suggests that such hands-on training can be usefully added to courses for training students and instructors that provide no other training in concurrency or training in declarative concepts. Our key ideas can be applied to languages other than Java.**

Keywords— fork, join, load balancing, synchronization, race conditions, automatic testing, hands-on training, Java threads

I. INTRODUCTION

Because of its importance, researchers are actively pursuing a variety of mechanisms for introducing parallel computing in undergraduate CS courses. One issue raised by this research is the kind of mechanisms used to implement concurrency – declarative mechanisms such as OpenMP [1] or procedural mechanisms such as the Java thread abstractions [2-5].

A declarative mechanism has the advantage that with little amount of code, it is possible to implement a variety of useful concurrent algorithms. The disadvantages are that it limits the range of supported algorithms, and more importantly, abstracts away the nature of a thread – an independent execution of a stack of procedure calls. This abstraction is particularly detrimental in a teaching training workshop [5], as the instructor would not be able to explain to the curious students the underpinning of concurrency concepts.

An alternative is to explore a hybrid approach in which both kinds of mechanisms are taught [5]. The disadvantage of this approach is the time required to cover the additional mechanism. This is a problem for both student and teacher training. Student training is often done by adding concurrency concepts to an existing course, which means sacrificing some existing serial concepts. Removing existing topics for adding one concurrency mechanism is difficult enough; doing so for two mechanisms is likely not practical. A similar problem occurs in a teacher-training workshop, in which instructors are gathered for a limited amount of time, which has to be devoted to both technical concepts and organizational and evaluation issues.

Given this problem, it is attractive to explore techniques to teach the basics of the procedural mechanisms in a time period that is typically affordable in a student or instructor-training course. To explore this idea, this work addresses the following specific question: Is it possible to introduce and test concepts in execution, forking, joining, synchronization, and load balancing of Java threads in one class period (which is approximately an hour) with TA help but without any lecture on this topic? If this goal is met, then such training can be usefully added to courses that provide no other training in concurrency or training in declarative concepts. Even if the goal cannot be met in its purest form, attempts to meet it can support weaker versions of the goal wherein the hands-on exercise usefully supplements recorded lectures, overview lectures, or even full lectures on the covered topics.

We believe this goal is ambitious given that we are using the class period for not only an introduction of five different concurrency topics, but also an evaluation of what the trainees learned. Therefore, we are not considering a shorter period such as 15 minutes. In a student-training course, such a period can be found, for instance, during a class period when a professor is away for a conference but has TA help. In a teacher-training course, such a period can be found, for instance, during a working lunch in which TA help is available.

In this paper, we describe and evaluate a technique we have developed to meet this goal and weaker forms of it. Section 2 describes previous techniques on which we build and identifies the primary key ideas we use to extend this work. Section 3 identifies additional secondary key ideas that address issues raised by our goal and the primary key ideas. Section 4 describes, in-depth, the hands-on exercise we have developed based on these key ideas, and our experience using this exercise in a class period when the instructor was away for a meeting. Section 5 gives conclusions and future work

II. RELATED WORK AND PRIMARY KEY IDEAS

The idea of lecture-less teaching [6] is not new, and has involved the use of recorded videos, textbooks, and tutorials. We have incorporated this general idea into the teaching of concurrent programming and used hands-on learning as a vehicle for implementing our approach. More important, we are

This work was funded in part by NSF award OAC 1924059.

considering, not a lecture-less course or module, but a single session to achieve all of our learning objectives.

Hands-on training is the typical vehicle used for teaching concurrent programming. However, previous work requires multiple class periods and involves some accompanying lectures, given just before or during the hands-on training sessions [1]. We have extended this idea to provide pure hands-on training, with no lectures, in one class period, which could be during the class or lab meeting time.

Two forms of hands-on training are possible. In one, trainees create new code from scratch [1]. In the other, they modify code given to them [5, 7]. Because of our goals of lecture-less, one-period training, we have adopted the second approach.

The code modification exercise has previously taken two forms. (1) Students uncomment and comment various parts of code [5, 7]. (2) Students extend the given code with new functionality [5]. Our innovation here is a third approach in which students fix bugs in the given code.

Another issue in code modification is how many different problems are given to the students to meet the learning objectives. Previous work involved many small problems [5, 7]. Given our goal of keeping the training session short, we chose to give them the code for a single problem.

Concurrent programs can be classified into those that (a) require concurrency to perform their task such as a program that simulates the dining philosopher problem, or (b) use concurrency to speed up a task, such as prime number detection, that could also be performed sequentially. As we are addressing HPC (high-performance computing), like most of the related work in concurrency pedagogy, we chose (b) and support a fork-join problem in which a dispatcher thread divides work among multiple worker threads it forks, and then joins them to receive their results.

Hands-on training has the potential to demonstrate the execution of multiple threads, and associated problems, in the user-interface, thus reducing the need for a more abstract lecture to do so. One approach to demonstrating concurrency is to create a graphical user interface in which multiple threads create animations that appear to execute concurrently [2, 3, 8, 9]. The second is to do so textually, in the console [5, 7], by showing interleaving of thread outputs that trace intermediate steps and final results. The first approach, typically, requires the use of a non-standard library or language for creating the user-interface. Moreover, HPC applications, in which concurrency is added to improve performance and not change the user-interface, are inconsistent with animations. On the other hand, the second approach has the disadvantage that console text must be manually mined by the trainee to identify concurrency. We use the second approach but keep the console text small by making the amount of work given to each worker thread small.

It is important that a trainee solution is not hardwired to specific data. Like exercises in previous work [1, 5], our problem makes the output a function of program arguments and random numbers generated by the program.

An important issue is the evaluation of the trainees' code. There may be none (typically in teacher-training courses), a manual evaluation, or automatic testing [5]. Automatic testing reduces evaluation time, and is particularly important if the code must be evaluated during the training session, as it reduces the time for the training session. In addition, when it checks functionality [5] rather than performance, it can serve as a vehicle to help guide trainees' work. Therefore, we use automatic functionality testing.

III. Secondary Key Ideas

The section above has identified several existing and new key ideas used in our work. In this section, we identify secondary key ideas to implement our goals and the primary key ideas identified above.

One issue specific to our idea of pure hands-on training is how to impart the knowledge needed to use the concurrency mechanisms. We leverage the fact that students are given working code that uses each of the concurrency mechanisms they are learning. We embed the code with comments describing how the mechanisms work. In addition, we incorporate, in the code, print calls that trace how these mechanisms work.

In Java, threads execute two kinds of procedures. The main thread, created by the Java virtual machine, executes the `main()` method. Other threads, forked by the main thread, execute a `run()` method in a Java Runnable interface. Unlike previous work in Java thread training [2-5], we explicitly bring out the relationship between these two kinds of methods.

Another issue specific to our goals is how to make sure a small amount of console text is used to demonstrate concurrent execution and associated synchronization problems, which requires the underlying system to either execute the threads concurrently or switch the execution of threads on a single CPU frequently. Since students run programs on their own computers, they are not guaranteed concurrent execution. Moreover, a CPU can execute small problems without doing any thread switching. We address this problem by, again, leveraging the fact that they are working on code given to them. In this code, we add calls to force thread switching.

Thread switching, by itself, is not sufficient to demonstrate synchronization problems as it also requires some unsafe shared accesses to occur during problem execution. It is possible and more efficient to create fork-join solutions that avoid such sharing of global data by using reduction techniques in which forked concurrent threads write to local data, which is read by the dispatcher thread after it joins them. We deliberately deploy the alternative approach in which the forked threads write to a shared data structure. We choose a problem that is likely to result in sharing even when worker threads perform a small amount of work. Finally, to ensure unsafe access, our code simulates register loads and stores in Java, and executes calls to trigger thread switching between the loads and stores. Thus, we deliberately include artificial code patterns in our code for illustration purposes. We did not have time, however, to explain how these patterns could be improved in a single class period.

It is typical for concurrency courses to include all of the code in the `main()` procedure [7]. We, on the other hand, use the separation of concerns software engineering principle, which says that independent tasks should be divided into separate classes and procedures. We create five classes: a (1) utility class,

which contains code the trainee is expected to use but not study such as the code to process main arguments; (2) worker class, which contains the Java `run()` method; (3) dispatcher class, which contains methods to fork and join methods; (4) repository class, which keeps the data shared among the worker and dispatcher thread, and (5) main class, which contains only the `main()` method. Each class performs a separate function and can be replaced by another class that performs the same function. More important, it can be understood on its own. This decomposition is problem-independent and can form a design pattern for Java fork-join problems, especially those used for hands-on training.

We put all of these classes in one Java file, for three reasons. First, this approach allows easier command-line compilation as a single file has to be compiled. Second, we can influence the order in which these classes are browsed, assuming trainees read them in the order in which they are placed in the file. Finally, a trainee can more easily search for prints in a single file.

Division of code in a class into procedures is driven by the principle that (a) each procedure with a bug should contain only the context needed to find and fix the bug, (b) each independent task in the fork-join model such as fork and join should be carried out by a separate method, (c) each thread should execute a stack of calls rather than a single call to illustrate it is associated with an independent stack, and (d) there should be multiple alternate ways of synchronizing methods in calls chains to remove race conditions.

To ensure everyone learns through the exercise, we do require TA support to help with it. To ensure that helped students learn the underlying concepts, we ask quiz questions that test this understanding. We exploit the fact that the output of our code traces the underlying algorithm by asking in the quiz what effect various modifications have on the output – both the ones they made to correct defects and others they could have made. This approach results in a new class of concurrency questions – output-based - to determine trainees' understanding of why code written by them or given to them works or does not. These questions, in turn, imply that the training period should, ideally, give them time to do both the coding part and answer quiz questions, when the code and output are still fresh in their heads.

IV. Summary of Key Ideas

We summarize below the primary and secondary key ideas we motivated in the previous sections:

1. Provide lecture-less training in the following concepts: concurrent execution, forking, joining, synchronizing, and load balancing of threads.
2. Let the coding part of this training consist of fixing bugs in a single program given to the trainees.
3. Let the program use concurrency to improve performance rather than provide thread-based functionality.
4. Implicitly demonstrate each of the concurrency concepts and bugs through a small amount of console output.
5. Provide tests, available during the exercise, for incrementally checking the correctness of bug fixing, and explicitly identifying symptoms of the bugs in the output.
6. Provide in-line and overview comments to explain the concepts needed to perform the tasks.
7. Demonstrate synchronization bugs in the output by using an artificial coding style involving pushing worker results to the dispatcher, simulation of register load and stores, and thread switching between a load and store.
8. Choose a problem that makes it likely that concurrent unsafe writes will be made to shared data.
9. Provide help to identify and fix bugs, while ensuring that output-based and other quiz questions check how much the trainees understood about the fixes they were helped with.
10. Use the separation of concerns principle to create the following class decomposition: a main class, a dispatcher class, a worker class, a shared-repository class, and a utility class.
11. Use the separation of concerns principle in method decomposition to ensure that: (a) each thread executes a stack of calls rather than a single call, (b) call chains can be used to identify alternate approaches for synchronizing methods, and (c) a buggy method does not contain any code that is irrelevant to its fix.

V. Realizing Key Ideas in Exercise

We have developed an exercise that realizes all of these key ideas. The author gave this exercise to students in a class when he was away for a meeting. Because of lack of space, we cannot give all aspects of the exercise and its execution. Below, we first describe the nature and composition of the class that performed the exercise, and their concurrency background. We then overview the specific goal of the program with which they worked. Next, we focus on the various tasks given to the trainees, which included both fixing code and answering quiz questions. We present (a) the specifics of the tasks, (b) overall statistics about how well the students completed them, (c) example answers to quiz questions, and (d) the components of the exercise that were designed to successively carry out the task. Finally, we present students' opinions about the exercise.

A. Nature of Class

The class was an upper-division course on programming language concepts. Its prerequisites included courses on object-oriented programming, computer organization, data structures, and automata theory. Almost all students were seniors.

The class had both in-person and Zoom participants, with about half the students attending in person. This was true also the day the exercise was given. Before the exercise was given to the students, the TA, and two students (who had taken a distributed thread-based course from the author) did the coding part of it. One of these two students and the TA were available for in-person help. The other student was available for remote help. No one asked for remote help.

Forty-two students were enrolled in the course when the exercise was given. Forty students, and the TA, submitted the result of their coding effort. Thirty-six students, and the TA, submitted quiz answers.

When asked in an earlier class about how much they knew concurrency, the ones who spoke, and the TA, said they knew almost nothing, which motivated the exercise. One of the survey questions asked quiz takers about their previous knowledge of concurrency, and the five specific topics: concurrent execution, forking, joining, synchronization, and load balancing of threads. 8% of the quiz takers had no background, 19% generally knew about concurrency, 49% had seen the specific topics but not their implementation, and 22% had seen implementations of the concepts, but not in Java. None of the answers illustrated below are from this 22%, who had more than a superficial knowledge of the covered concepts.

All previous classes had live lectures, in which students' class participation was evaluated based on how many questions they answered posed through Zoom chats. They were told that in the exercise-based class, the quiz grade would be worth roughly one day's Zoom class participation, and the coding part about double the points allocated to a Hello World program they wrote to get familiar with the testing and other infrastructures used in the course. Thus, this exercise was designed to be a low-stake, high-reward one. They were given a week to make final submissions.

28 students simultaneously opened the assignment Google Docs document during the class period (The author was monitoring the number remotely). The rest, we assume, used other deadlines to postpone this work. This document was shared with them 15 minutes before class started. 23 students submitted their final version of the tested code within about 90 minutes of when the 75-minute class started, and eleven of them within 50 minutes. Nine of them submitted their final versions of the quiz answers within about 90 minutes. Many more had made their initial quiz submission within this period. However, the author encouraged them to revisit a couple of questions, including a survey question, that had not been interpreted correctly by many. Had the submission deadline been stricter, we believe the vast majority would have finished both the coding and quiz tasks within 90 minutes, before the next class that day started.

This was a good trainee sample to test this first-cut effort at such an exercise as the class had advanced undergraduate students, and the majority of them had not seen the specific concepts covered, in theory or practice. This exercise was intended for not only training students but also instructors, and the seniors were in between the two groups in maturity.

B. Overview of Exercise

The assignment write-up and the commented program formed the entire exercise. The functional goal of the program given to them was to take as an argument an integer N, generate an array of N random numbers, find the odd numbers in this array, and print a list of the odd numbers as well as the total number of odd numbers. The algorithmic goal was to use the fork-join model to implement this functionality.

The code given to students partly met these goals, and their coding task was to fix it. We chose identification of odd numbers rather than, say, the more compute-intensive task of finding prime numbers, as the likelihood of a random number being odd is high, which in turn, makes high the likelihood of concurrent accesses to the data structure that keeps track of the results. The assignment write-up explained the two program goals.

Two JUnit tests were written for the exercise, which were identical except that they gave different arguments to the `main()` method. Students could run them locally on their computer and also on the Gradescope server. They had to make their final code submission to Gradescope. The quiz was taken and submitted on Gradescope.

With this background, we are now ready to describe their tasks, which involved fixing bugs and answering quiz questions.

C. Forking Threads

We start with a quiz question, not because it was their first task, but because it checks their understanding of the most fundamental concept they needed to perform the other tasks – concurrency and thread forking. The question:

Assume thread p executes the following code, where r is a Runnable:

```
Thread t = new Thread(r);
r.run();
```

Explain why this code will make thread p execute `r.run()` *rather than thread t. If you are unsure why, execute this code with the Runnable implemented for you, put a breakpoint on run(), and see the stack. How would you change this code to make thread t execute* `r.run()`*?*

The following sample answer provides an expected response: "The stack for the thread t has not been created therefore the code will run on the stack for thread p. To make the thread t execute r.run() we would need to call t.start() to start the thread represented by t."

8% of the quiz takers (3 students) skipped this question. 76% answered it correctly. The mistakes were in suggesting, as the fix (1) `t.run()` (11%), or (b) `t.start()` followed by `r.run()` (5%). We believe the first mistake was caused by the fact that "running" and "starting" a thread are synonymous terms in English; the author has seen many students confuse the two, which was the reason for this question. The second mistake was probably caused by the students correctly thinking of thread forking as a two-step process in which the: (1) thread first starts, and (2) then the Runnable runs. They did not realize that the thread `start()` call performs both functions. One student asked a TA to explain the concept of concurrency and how it was relevant to the exercise, and thus, this help may have contributed to the answer to this question.

The exercise contained at least two components to answer this questionThread starting was illustrated in the code given to them, which would explain their success in answering the second part of the question. For the first part, comments (Fig. 1) described `main()` and `run()` as the root methods of stacks created by Java and the programmer, respectively.

In general, a thread is an independent unit of steps, coded in its root method. The `main()` method is always at the root of the thread's stack, and thus, is called the root method of the thread. Other methods such as the dispatcher methods get pushed on top of the stack when they are called and popped when they return. A thread starts when its root method is called and terminates when the root method ends. In between the start and stop, a stack of calls can be serviced by a thread. This stack grows and shrinks, as different methods are called. In this example, for instance, some example stack snapshots are:

```
ConcurrentOddNumbers.main()
->OddNumnbersUtil.fillRandomNumbers
->OddNumbersUtil.generateRandomNumbers
ConcurrentOddNumbers.main()
->ConcurrentOddNumbers.fillOddNumbers()
```

a) Comments identifying a thread as an independent stack

The call `t.start()` starts the thread represented by t, that is, creates a new stack, and executes the `run()` method of the Runnable instance bound to t

b) Comments explaining thread start

Figure 1. Comments explaining threads

D. Joining Threads

The students were told that the method in Fig. 2 was buggy and asked to fix it. Three students needed help with identification of the bug. In the words of the TA, he "described/tried to get them to describe what was happening with the loop (fork thread 1 -> join thread 1 -> fork thread 2, etc.) and then asked them what they thought it should be doing. They all replied with fork all -> join all, and were able to figure out the problem and solution from there."

```
private static void forkAndJoinThreads () {
  for (int aThreadIndex = 0;
           aThreadIndex < threads.length;
           aThreadIndex++) {
     forkThread(aThreadIndex);
     joinThread(aThreadIndex);
   }
}
```

Figure 2. Fork-join bug.

The traced output, shown in Fig. 3 had clues to identify and fix the bug. To determine if the students could see these clues, the following quiz question was given: *How does the traced output of the original buggy version of* `forkAndJoinThreads()` *differ from that of the corrected one?*

Some answers to the question correctly identified the problem with the output; for example: "The thread outputs are not intertwined. They appear sequentially instead: The later thread begins after the former thread provides output." Others described the problem with the code rather than symptoms in the output; for example: "The problem is that different threads in the original codes run sequentially instead of concurrently." Either way, they understood the problem with the method. The tests further helped them identify the problem, as illustrated by the message shown in Fig. 4. The result was that everyone successfully fixed the bug.

E. Load Balancing

To introduce the load-balancing bug, we created two methods, shown in Fig 5 to allocate the number of random numbers each thread would process. Students were told that the bug was in the second method, `fairThreadRemainderSize()`, which was called by the first method, `threadProblemSize()`.

This problem caused the most coding difficulties, perhaps because of the artificial division of work allocation into two methods to create the bug. Each TA fielded questions from four students, and one student needed help twice. One student did not realize two methods were involved in the computation and they needed to use the value of the `aThreadIndex` argument. Based on hints given to them, most of them realized that the return value should be between 0 and 1, and then proceeded to fix the solution. All but one of the students passed both tests for this bug. Recall that these tests generate different number of random numbers. One person passed only one test, probably because of hardwired return values for one of these tests.

```
1.  Thread 1->Starting:Thread 13
2.  Thread 1->Stopping execution until the following
    thread terminates:Thread 13
3.  Thread 13->run() called to start processing
    subsequence:0-3
4.  … // Thread 13 output
5.  Thread 13->run() terminates to end processing of
    subsequence:0-3
6.  Thread 1->Resuming execution as the following
    thread has terminated:Thread 13
7.  Thread 1->Starting:Thread 14
8.  Thread 1->Stopping execution until the following
    thread terminates:Thread 14
9.  Thread 14->run() called to start processing
    subsequence 3-4
10. … // Thread 14 output
11. Thread 14->run() terminates to end processing of
    subsequence:3-4
12. Thread 1->Resuming execution as the following
    thread has terminated:Thread 14
```

Figure 3. Output identifying fork-join and load-balancing bugs

```
Forked threads do not execute concurrently.
Between the first and last output of each forked
thread, there is no other thread output.
```

Figure 4. Test error message identifying fork-join bug

Except for one student, all of the helped students had made at least some attempt at a solution when they asked for help, indicating that they had likely identified the bug. As with the previous problem, both the output and test results indicated the issue. In Fig. 3, line 3 shows that Thread 13 was allocated sequence 0-3, and line 9 shows that Thread 14 was allocated sequence 3-4. Other lines, not included here, indicate that all other threads were allocated one element. Test output, shown in Fig. 6, explicitly indicates the problem and implies that the problematic method should return a value between 0 and 1. This hint could have been explicitly given, but since we had TA help available, we decided students would learn more if they figured this out themselves. The vast majority did.

It is possible that some students did not run tests to determine the problems; instead, they waited until they were satisfied that their program was bug-free, and then ran the tests. Recall that these tests could be run locally, as part of a library called LocalChecks, or as part of the Gradescope server, into which the library and other code was uploaded. Because of constraints of the Gradescope server, some error messages produced by the tests were suppressed on the server. One of the TAs' mentioned that: "1 student was not running the LocalChecks grader and was instead attempting to directly run grading in Gradescope. This was fine, but he was not getting any error messages, and so, he could not figure out what was wrong with his code. I told him to run LocalChecks, and that displayed a nice error message. After that, he didn't have any more issues." The test outputs seemed to have helped at least this student. All error messages shown here (Figures 4, 6, and 9) were produced by the LocalChecks library.

The following quiz question addressed load-balancing: *How does the traced output of the original buggy version of* `fairThreadRemainderSize()` *differ from that of the corrected one?* All students demonstrated their understanding of the problem, though many, like the answer below, did not explicitly indicate how the output changed: "The original version placed the entire remainder on the first thread causing it to take about half of the load (in small sample size) while the other threads only handled one index. The corrected version only had at most a one index difference in the load of each thread."

F. Synchronization

The synchronization mistake was in the method shown in Fig. 7, which simulates a register-based increment to a shared variable. In this case, students were told how to fix it – they simply had to uncomment the synchronized keyword in the first line and observe how it influenced the output. This led to one student asking if that was all that was involved!

The intellectual exercise here was to answer three questions on this change. The first*: In what ways does the traced output of the original unsynchronized version of* `incrementTotalOddNumbers()` *differ from that of the corrected synchronized one? Assume in both cases that* `forkJoinAndThreads()` *has been corrected. Consider differences in the output of both the final and intermediate results - in particular, the values loaded and stored.*

Some answers (27%) explained both the change in final and intermediate values. The following is an example: "In the original unsynchronized version, the traced output would likely showcase inconsistencies and race conditions, where multiple threads may load the same initial value of `totalnumberoddnumbers`, leading to incorrect increments and inaccurate intermediate and final results. However, after I corrected the synchronization, it would ensure that the threads access the method sequentially, preventing concurrent access and modification, which would reflect in the traced output showing consistent and correct final values". The most common answer (41%) explained only the change in final values, while some (27%) explained what the problem and solution were, without mentioning symptoms in the output. All answers demonstrated an understanding of synchronization. One person did not answer the question, and one flipped the two cases.

```
/**
  * This method determines how many elements of the
  * input list, whose size is, aProblemSize, will be
  * processed by the thread whose index in the
  * thread array is aThreadIndex.
  */
private static int threadProblemSize(
        int aThreadIndex, int aProblemSize) {
  // Following is the size if the problem can be
  // evenly divided among the threads
  int aMinimumProblemSize =
        aProblemSize / NUM_THREADS;
  // This is the remaining work
  int aRemainder = aProblemSize % NUM_THREADS;
  return aMinimumProblemSize +
     // calculate out how much of the remaining
     // work is done by this thread
     fairThreadRemainderSize(
        aThreadIndex, aRemainder);
}
/**
  * The goal of this method, as its name suggests,
  * is to divide aRemainder items fairly among the
  * available threads, that is, the differences in
  * the sizes of the portions is as small as
  * possible. aRemainder is expected to be between
  * 0 and NUM_THREADS - 1;
  */
private static int fairThreadRemainderSize(
  int aThreadIndex, int aRemainder) {
  if (aThreadIndex == 0) {
    return aRemainder;
  } else {
    return 0;
  }
}
```

Figure 5. Load-balancing bug

```
Imbalanced thread load: Max thread iterations(4)
- min thread iterations(1) = 3. It should be <= 1
```

Figure 6. Error message identifying load-balancing bug.

```
// synchronized
static void incrementTotalOddNumbers() {
  int aRegister = totalNumberOddNumbers;
  // Simulate load memory to register
  printProperty("Loaded total number of odd
numbers", totalNumberOddNumbers);
  // Before the incremented register is saved to
  // memory, another concurrent thread may also load
  // the same value for totalNumberOddNumbers in its
  // local register variable.
  aRegister++; // increment register
  ThreadSupport.sleep(10);
  // The above call simulates a CPU switching
  // execution to another thread.
  totalNumberOddNumbers = aRegister; //save register
  printProperty("Saved total number of odd numbers",
totalNumberOddNumbers);
}
```

Figure 7. Synchronization bug and solution.

Fig. 8 contains the output trace showing the problem with unsynchronized access. Between the loads and stores are elided outputs for thread starts and other events, which may have been

missed by some of the students who did not mention it. The test result in Fig. 9 indicates a mistake in the final result.

The following question required an answer that could not be determined directly by examining the output or test results: *Suppose we execute the initial buggy version of* `forkJoinAndThreads()`. *Explain why making* `incrementTotalNumbers()` *synchronized results in no difference in the output. That is, under the condition above, whether the method is synchronized or not has no influence on the traced output. Feel free to examine the output under these conditions to help understand why.*

```
Thread 14->Loaded total number of odd numbers:0
… //more output
Thread 13->Loaded total number of odd numbers:0
..// more output
Thread 13->Saved total number of odd numbers:1
..// more output
Thread 14->Saved total number of odd numbers:1
```

Figure 8. Output demonstrating synchronization bug.

```
Computed total number of odd numbers 1 != expected total 3
```

Figure 9. Test message identifying synchronization bug.

86% of the students gave a completely correct answer. An example: "Since each thread was running consecutively rather than concurrently, the counter was only being accessed by one thread at a time, which is the same as if `incrementTotalNumbers`() was synchronized." Three gave a partly correct answer, and two gave incorrect answers. As expected, this was a harder but answerable quiz question.

The following turned out to be the most difficult question, answered correctly by only 46% of the students: *The only caller of the static* `addOddNumbers()` *method is the instance run() method, and the only caller of* `incrementTotalNumbers()` *is the static* `addOddNumbers()`. *Suppose* `addOddNumbers()` *is not synchronized but* `run()` *is synchronized. Explain why in this scenario it matters whether* `incrementTotalNumbers()` *is synchronized or not. Look at the explanation of synchronized methods in the comments above* `incrementTotalOddNumbers()` *if you are not clear about the reason.*

Example correct answer: "If `run()` were to be synchronized, it would lock the instance, since `run()` is an instance method and not a static method. This means that other threads in unlocked instances can still execute `run()`, but since `incrementTotalNumbers()` is not also synchronized, it does not function in the desired way. Multiple instances would be executing the `run()` method, which would eventually have a call to `incrementOddNumbers()`, and without the synchronization, the bugs discussed in Q3 would occur."

The overview comments did explain that executing a synchronized static and instance method locked the synchronized class and instance methods respectively. We believe that 54% of the answers did not explain this correctly because there was no exercise experiment to illustrate this difference, reinforcing the idea of hands-on training.

G. Survey on Exercise Impact

The "quiz" contained two post-exercise survey questions.

The first asked the importance of the *new concepts* they learned through this exercise that they did not know earlier. The responses were: 5% not important, 32% moderately important, 38% very important, and 24% so important that no one should graduate without knowing them.

In their explanations, some seemed to giving the importance of the topics rather than what they had learned. Therefore, the author clarified the question in a forum post, and asked them to edit their answers, if necessary. It is not clear if all who had this confusion edited their answers. However, some were clear, as illustrated by the following answer: "Moderately important - I've known the concepts before, but I found it helpful to see how they are implemented in code."

An interesting rationale for a tepid response was: "Moderately important, so far throughout all the computer courses I've taken, most of the assignments I've done don't involve coding threads and runnables. I think the knowledge on threads can be useful in regards to efficiency in running programs." An interesting rationale for an enthusiastic response: "Because I have been asked about designing a concurrent program during an interview. It is important in modern industry." Together, these responses motivate more exposure to concurrency concepts in a wide range of CS courses. The overall sentiment was very positive, most enthusiastically expressed by the following explanation of a "very important" rating: "This rocked, thanks."

The second survey question asked them if, given a choice, they preferred this exercise for these concepts or the kind of interactive lecture with Zoom-based Q/A they had experienced in this course in earlier classes. In this form of Q/A, the author posed a question in Zoom chat, waited for the average student to compose an answer, asked students to hit Enter, and then discussed the answers.

The responses were: 24%, lecture, 22% not sure, 14% little difference, and 38% exercise. The reasons for preferring a lecture included: It gives better explanations, serves better as an introduction, is more visual, promotes more thinking (through the Zoom Q/A), allows learning from peer responses, answers important conceptual questions more, and does not require time after class to finish the work for those who code at a slower pace. Some of these responses implied such an exercise was effective as long as it was a one-time occurrence, which is what it was intended to be.

The reasons for preferring the exercise included: It is a more active form of learning, provides more depth, is tougher, is more self-paced, does not require the fast student to wait for the average student to provide Zoom answers, does not require quick public Zoom responses, and gives reproducible examples.

A reason given for the response of (a) "little difference" was that the exercise was guided like a lecture and thus as effective, and (b) "not sure" was the desire to see a hybrid model with a video introduction to the exercise.

The responses to these two questions indicate that, overall, the exercise was an attractive alternative when a professor is not available for an in-person lecture, and, despite limitations perceived by some, made at least a moderate impact on the learning of those who had seen some of these concepts before and those who had not.

VI. Conclusions and Future Work

The main contribution of our work is to consider, in-depth, the possibility of a lecture-less introduction, in an hour, to imperative mechanisms for concurrent execution, forking, synchronization, and load balancing of threads. Not all students finished all the tasks in an hour, but many did. Students did get all the help they needed within the hour, and were able to finish the quiz questions on their own later. Finishing later is consistent with the fact that in a university course, a student is expected to work 3 hours outside class for each hour of class. The TA, who had little knowledge of concurrency, took 15 minutes to read the assignment write-up and get the code set up, 15 minutes to look through all of the code and associated comments, 20 minutes to fix the bugs, and 5 minutes to check the work and submit to Gradescope. If the exercise is published ahead of time, then the vast majority of trainees may have been able to finish both the coding and quiz within an hour.

The fact that (a) all students submitted correct code, with some requiring help, and (b) the vast majority of students answered the quiz questions correctly, with some making understandable mistakes, indicates that the covered concepts were nontrivial and learnable through a pure hands-on exercise.

The design of a lesson for a live lecture can leave many aspects to be resolved at lecture time, as a misstep can be corrected during the lecture. The design of a pure hands-on session does not have this luxury, even with TA support, as such support provides 1-1 help, and thus, does not scale. In comparison to papers on lecture design, this work addresses the "devil in the details" more thoroughly, and thus provides a more comprehensive description of its approach, parts of which could be reused in several future endeavors. Its evaluation is equally detailed, giving specific problems fielded by the TAs, the hints given to students, and the mistakes they made in quiz questions, to identify the issues that may need to be addressed by future offerings of the exercise.

It would be useful to investigate the applicability of this exercise in a variety of settings, including (a) teacher-training workshops, (b) courses addressing OMP programming that are taken by students with Java familiarity, and (c) courses on Java-based object-oriented programming. Often a professor has to be at a conference during lecture hours – this exercise is an attractive alternative to a lecture given by a guest teacher. We will be happy to share the assignment write-up, downloaded code, local checks library, and the Gradescope autograder with others interested in using this exercise.

It is even more attractive to consider variations of this approach. An introductory YouTube lecture could introduce Java threads, while still keeping the lesson free of a live lecture. This step could reduce the time required to do the exercise, as reading the concept-explaining comments in the code (e.g. Fig. 1) may not be necessary. Moreover, the exercise can be made easier in various ways, while still serving its objective of providing a hands-on introduction to the basics of the covered concepts. For instance, in the faulty load-balancing method, a comment can be added to indicate that the return value should be between 0 and 1, an implication that was derived with TA help in some cases. Similarly, print statements can be added in this method to identify the return value. Further, the quiz questions can be omitted.

Conversely, there are many ways to go beyond these basic concepts. With more TA or automated help, the clues about the problems in the output, comments, and test results could be incrementally given, as needed by the trainees. Identification of which methods are buggy could be the responsibility of the trainees unless they ask for this information. Thread switching could be introduced in the middle of the list operation to add an odd number to the final list, creating additional synchronization problems. The static shared data could be manipulated in a synchronized `run()` instance method, making the trainees responsible for moving this code to one or more static synchronized methods. Trainees could be told to use reduction techniques to avoid the need for synchronized methods. These additions could be addressed in out-of-class time and should. not exceed the expected 3 hours.

Even more attractive is to adapt the exercise to support imperative training in other languages such as C, C++, and Python. While the concrete realization of it would change in these adaptations, all key ideas summarized in Section IV except class decomposition, would apply. Class decomposition would also apply to other object-oriented languages such as C++ and Python. The biggest challenge in these adaptations would be implementing concurrency tests. We built our tests using the functional testing capabilities of a Java-based testing infrastructure [5]. Similar infrastructures, developed for other languages, could support such training in these languages.

This paper provides a basis for pursuing these exciting directions.

Acknowledgment

Thanks to Mason Laney and Felipe Yanaga for identifying the trainee problems they addressed and the hints they gave.

References

[1] Ghafoor, S., D.W. Brown, and M. Rogers. Integrating Parallel Computing in Introductory Programming Classes: An Experience and Lesson Learned. in Proceedings of the Euro-EDUPAR 2017.

[2] Bruce, K.B., A. Danyluk, and T. Murtagh, Introducing Concurrency in CS 1, in Proc. ACM SIGCSE'10. 2010, ACM.

[3] Dewan, P., S. George, A. Wortas, and J. Do, Techniques and tools for visually introducing freshmen to object-based thread abstractions. Journal of Parallel and Distributed Computing, 2021. 157.

[4] Ricken, M. and R. Cartwright. Test-First Java Concurrency for the Classroom. in Proceedings of the 41st ACM SIGCSE. 2010.

[5] Dewan, P., A. Worley, S. George, F. Yanaga, A. Wortas, J. Juschuk, M. Rogers, and S.K. Ghafoor, Hands-On, Instructor-Light, Checked and Tracked Training of Trainers in Java Fork-Join Abstractions., in HiPCW. 2022, IEEE. p. 28-35.

[6] Zanca, N.A., Lecture vs. Lecture-less: A Meta-Analysis from Journal of Economic Education (1969 to 2016). Journal of Economic Insight, 2017. 43(2).

[7] Adams, J.C., Injecting parallel computing into CS2, in Proceedings of the 45th ACM SIGCSE. 2014, ACM: Atlanta, Georgia, USA. p. 277-282.

[8] Lönnberg, J. Defects in concurrent programming assignments. in Proceedings of the Ninth Koli Calling International Conference on Computing Education Research (Koli Calling 2009). 2010.

[9] Bogaerts, S., Hands-on Parallelism with no Prerequisites and Little Time Using Scratch, in Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses, S. Prasad, A. Gupta, A. Rosenberg, A. Sussman, and C. Weems, Editor. 2019,