Hands-On, Instructor-Light, Checked and Tracked Training of Trainers in Java Fork-Join Abstractions

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

Andrew Worley
Department of Computer Science
Tennessee Tech University
Tennessee, USA
apworley42@students.tntech.edu

Samuel George
Department of Computer Science
University of North Carolina
Chapel Hill, USA
sdgeorge@cs.unc.edu

Felipe Yanaga
Department of Computer Science
University of North Carolina
Chapel Hill, USA
yanaga@unc.edu

Andrew Wortas
Department of Computer Science
University of North Carolina
Chapel Hill, USA
ajwortas@cs.unc.edu

James Juschuk
University Relations
IBM Corporation
Research Triangle Park, USA
jushchuk@ibm.com

Mike Rogers
Department of Computer Science
Tennessee Tech University
Tennessee, USA
MRogers@tntech.edu

Sheikh Ghafoor
Departmemt of Computer Science
Tennessee Tech University
Tennessee, USA
SGhafoor@tntech.edu

***Abstract*— As part of a 3-day workshop on training faculty members in concurrency, we developed a module for hands-on training in Java Fork-Join abstractions that had several related novel pedagogical and technical components: (1) Source and runtime checks that (a) tested whether test-aware code created by the trainees met the expected requirements and (b) logged their results in the local file system and the IBM cloud. (2) Editable worked example code along with a guide on how to understand the underlying concepts behind the code and experiment with the code. (3) The ability to follow the guide (a) synchronously, with graduate student help, in a session devoted to this module, and (b) asynchronously, on one's own, before or after the synchronous session. (4) Assignments trainees could do after experimenting with the worked example. (5) Zoom recording of the entire synchronous session. Fourteen faculty members across the country attended the session and had varying amounts of knowledge of Java and automatic assessment. Data gathered from check logs and a Zoom recording, together with novel visualizations of them, provide information to evaluate our pedagogical model and differentiate the participants.**

Keywords*— *awareness, instructor dashboard, automatic grading, automatic help, testing, metrics, concurrency, education, hands-on learning

I. Introduction

Because of its importance, researchers are actively seeking methodologies and tools for introducing PDC (Parallel and Distributed Computing) in introductory CS courses. In [1], the authors present their effort to implement parallelism in first and second-year CS courses. The authors found that students were capable of learning the material and enjoyed the experience. In [2], the author suggests that a data structure course is a natural place to introduce parallelism, while several researchers have focused on teaching PDC topics to students in upper-division courses [3] [4]. Our previous work has motivated integrating the teaching of object-oriented programming and concurrency [10, 11]. Researchers have also attempted to integrate PDC throughout the curriculum [5-9].

These instruction efforts have been led by leaders in concurrency pedagogy. The impact of these efforts is limited by the concurrency training of potential instructors of concurrency. The last author sent a survey to CS faculty of over a thousand four-year institutions across all fifty states in the US, and one hundred and thirty-five responses were received. Nearly 60% of the respondents indicated that they would not integrate PDC topics or would be unlikely to integrate PDC topics into their curriculum without further training or resources that can be readily integrated.

How this training should be imparted to potential instructors is a first-class issue in its own right, separate from the issue of how PDC should be incorporated into the curriculum. In theory, these two issues could be integrated if every untrained (in concurrency) instructor could take, as a student, one or more PDC courses/modules taught by trained instructors at the same or different institution. However, this approach is impractical for at least four reasons. First, the untrained instructors have to be committed to teaching concurrency. Second, and more important, they need to find enough time in their schedule to take such courses. Third, most universities open their courses only to enrolled students. Finally, the instructors might lack background in the programming language/environment used in the course.

To circumvent these problems, PDC training to potential instructors on various topics has been limited to conference sessions focused on a specific topic or special workshops addressing a wide range of topics. Due to the nature of the material, such sessions should, ideally, be hands-on, involving concurrent programming. Based on our experience observing and conducting these sessions, we can define the following model for them.

The session is devoted to a main topic, such as OpenMP in C. The topic is broken into subtopics such as different Open/MP directives. For each subtopic (such as the parallel directive), a university professor, typically a leader in the field, gives a conceptual presentation, showing worked examples (such as a parallel hello world), and then gives the participants one or more exercises that use the concepts associated with the topic. The participants may work alone or in groups. The professor with the help of possibly some graduate students helps those who articulate their problems. After the allotted time for the topic, the instructor moves on to the next subtopic.

This work was funded in part by NSF awards OAC 1829752 and 1924059.

.

This approach has two main problems. First, it is instructor-heavy, because, as mentioned above a leader in the field gives the presentation, which limits its scalability. Second, no mechanism is used to evaluate the pedagogy or differentiate the participants. In the allotted time, it is possible for the average participant to finish much earlier or not at all. Worse, given that many programmers are shy about asking for help [12-14], some participants may not make any progress with the problems. It is perhaps because of a lack of data about the effectiveness of these sessions that, to the best of our knowledge, there is no paper on training of PDC trainers.

In this paper, we motivate present and evaluate a new approach to address these two problems. It has several novel technical and pedagogical components: (1) Source and runtime checks that (a) test whether test-aware code created by the trainees meets the expected requirements and (b) log their results in the local file system and the IBM cloud. (2) Editable worked example code along with an experiment-based guide on how to unravel the underlying concepts. (3) The ability to follow the guide (a) synchronously, with graduate student help, in a session with a time limit, and (b) asynchronously, on one's own, before or after the synchronous session. (4) A set of assignments trainees can do after experimenting with the worked example. (5) Recording of the synchronous session.

In Summer 2022, as part of a 3-day workshop on training faculty members in concurrency, we applied this approach to hands-on faculty training in Java Fork-Join abstractions using a 100 minute synchronous session. Fourteen faculty members across the country attended the session. They had varying amounts of knowledge of Java and automatic assessment. Check logs and a zoom recording give a detailed picture of the model and differentiate the participants along several dimensions.

Section II outlines the context for our work Section III clarifies what we mean by fork-join. Section IV gives our pedagogical model. Section V evaluates our approach using th novel visualizations of data extracted automatically from the check logs and manually from the Zoom recording. Section VI presents conclusions.

II. Context of Training Session

The context of our work was a faculty-development workshop organized by the Computer Science Department at Tennessee Tech (TTU) in collaboration with the CDER center in the summer of 2022 The workshop was targeted at faculty who usually teach (or are scheduled to teach) early programming classes and who do not have parallel and distributed computing expertise. The goal was to help them integrate PDC into these courses. The workshop was funded by NSF (National Science Foundation). Selected participants were paid stipends.

The workshop exposed participants to concurrent programming using C/C++, Java, and Python; and C and Java OpenMP processors. It also introduced them to PDC-related pedagogy and assessment techniques. It had both a virtual component and an in-person component. The in-person component consisted of three day-long faculty development activities at Tennessee Tech University.

Each technical session of the workshop followed the traditional model described in Section 1, that is, lectures by experts interleaved with hands-on programming activities. Participants were separated into four groups with graduates available to provide assistance. For all but one session, there was no assessment. The exception was a session on the last day that used our pedagogical model to explain Java fork-join. The night before the session, the participants were emailed the guide. Those who were able were encouraged to asynchronously download the guide, checks, and scaffolding code, and even start following the guide.

At the start of the session, the first author gave a remote virtual (Zoom) 50-minute interactive talk in which he first surveyed the background of each participant and their reason to attend the workshop, and then motivated and explained the idea of automatic assessment. The survey asked each participant the following three questions:

1. *Have you ever taught a Java-based course?* Ten participants responded "yes" to the first question, with one of them saying they had taught Java only once and another saying their experience was mostly in C++.
2. *Have you given an assignment that has been at least partially auto-graded?* Two had used Zybooks, one Peerson, one Senagauge, and one an unnamed system.
3. *What is your interest in concurrency?* There were five responses: (1) Get practice in parallel algorithms, (2) Collaborate with CS department to teach an HPC course, (3) Understand cluster computing and teach concurrency to students, (4) Guide Capstone projects and understand Machine Learning systems. (5) Teach freshmen as they need it in upper-level courses.

Thus, we see the participants had varied motivations for and backgrounds in the material covered. Four had not never taught Java, and two were not fluent in it. Nine out of fourteen had never used any kind of auto-grader, and the remaining five had used four different kinds of auto grading systems.

After the motivational talk, the participants were given about 100 minutes to follow the guide. They worked in the same four groups in which they worked in other sessions. Each participant made changes to their individual code base, and consulted other members of the group and three TTU graduate students when they needed help.

III. Fork-Join

The term fork-join is applied to both algorithms and programming abstractions. As we see below, these are related but different concepts.

A. Fork-Join Algorithm Model

While fork-join abstractions have well-defined semantics, to the best of our knowledge, fork-join algorithms have no well-defined definition. Such a definition is crucial to describe our pedagogical model. Moreover, it forms a basis for the implementation of our source and runtime checks.

We assume that a program following the model consists of a single *dispatching thread* and `M` *worker threads*, **`M`** >= 0. The dispatching thread creates or *forks* the worker threads and then *joins* all of them, that is, waits for all of them to finish. Before forking threads, it performs **F** *pre-fork* actions, and after joining, them it performs J *post-join* actions, **F** and **J** >= 0. Each worker thread $\mathbf{W}^{\mathbf{i}}$, i <= i <= **M**, performs an interactive loop B number of times, $\mathbf{B}^{\mathbf{i}}$ >= 0. Before entering the loop, it performs **A** pre-iteration steps and **C** post-iteration steps, **A** and **C** >= 0.

Thus, the control flow for the dispatching thread is: (1) Perform **F** pre-fork actions. (2) Fork **M** worker threads. (3) Perform **J** post-join actions. The control flow for each worker thread $\mathbf{W}^{\mathbf{i}}$ is: (1) Perform **A** pre-iteration steps, (2) Perform $\mathbf{B}^{\mathbf{i}}$ iteration steps, (3) Perform **C** post-iteration steps. The number of iteration steps is thread-dependent because work cannot always be divided evenly among threads.

TABLE I. FORK-JOIN PARAMETERS

F	Number of pre-fork actions by dispatcher thread
M	Number of threads forked by dispatcher thread
J	Number of post-join actions by dispatcher thread
A	Number of pre-iteration steps by each worker thread
$\mathbf{B}^{\mathbf{i}}$	Number of iteration steps by the ith worker thread
C	Number of post-iteration steps by each worker thread

B. Java Fork-Join Abstractions

There are many ways in Java to fork and join threads, some higher-level and/or more stylistically correct than others. For instance, to fork a thread, it is possible (a) to create a class that implements the Java `Runnable` interface and pass an instance of it as a constructor argument to the Java `Thread` class, or (b) to subclass the Java Thread class directly and override its `run` method. The former is more stylistically correct – hence we assume this approach. Similarly, joining of worker threads can be supported (a) by the dispatching thread making Java `wait` calls and workers making `notify` calls, or (b) by the dispatching thread making a `Thread join` call on each worker thread. The latter is higher level as it does not require explicit synchronization between the worker and dispatcher threads – hence we assume this approach.

C. Fork-Join Testing Requirements

In our previous work, we developed an approach for testing concurrency requirements [15] for applications that visualize their concurrency [10, 11]. Our fork-join algorithm model assumes a variable **M** and thus does not guarantee such visualization. When **M** = 0, the multi-threaded version is identical to the single-threaded one. This means that the input-output relationships of single-threaded and multi-threaded are the same.

To overcome this problem, we require a testable program to produce special output to demonstrate its concurrency steps. The program must produce special output to identify the nature and number of the various kinds of steps We specify to each runtime check the values of **F**, **M**, **J**, **A**, $\mathbf{B}^{\mathbf{i}}$, and **C** and regular expressions describing the output produced by these steps. The check ensures that the program follows these requirements while also checking that work is balanced among the threads. As this output would not be required if the program was not automatically tested, we call a program that produces such output as *testing-aware*. However, as we see later, such output also has the potential pedagogical benefit of helping programmers understand the concurrency behavior of their implementations and also to debug their implementations.

IV. PEDAGOGICAL MODEL AND IMPLEMENTATION

Our guided active presentation serves as a general pedagogical model for explaining how the fork-join algorithm model should be implemented using the Java fork-join abstractions. We refer to the implementation of the general model in the TTU workshop mentioned as simply the implementation. The implementation in our guide is 26 pages long (with figures). Here we describe the underlying model by identifying and illustrating the principles it uses including a focus on multithreading, a common programming pattern for implementing fork-join, in-context concept descriptions centered around source and runtime checks, and the use of the underlying programming environment and tests to unravel concepts through guided experimentation,

A. Focus on Multithreading

Our pedagogical model assumes implementations of single-threaded versions of assignments are given to the participants as scaffolding code. In our implementation, the two planned assignments were to find, using four threads, (1) the prime numbers in a set of random numbers and (2) the value of PI using the Monte Carlo method. The participants were told that the PI problem was optional but they should try and solve the prime number problem, time permitting. Not requiring participants to implement the single-threaded aspects of the problems allows us to focus more on the common concurrency aspects of the problems rather than idiosyncrasies of single-threaded implementations, which is particularly important when a single session is devoted to the matter.

In addition, we assume a partial implementation of a separate problem, we call a worked-example, that correctly implements the single-threaded requirements and also has all of the code, some of it commented out, for meeting the multi-threaded requirements. In our implementation, the worked example was to find the odd numbers in a set of random numbers, also using four threads. The code to implement multi-threading is commented out, so the uncommented code only does single-threading. The guide asks the participant to incrementally uncomment the multi-threaded code and observe the results to understand the steps it performs,

B. Fork-Join Programming Pattern

The implementations of the worked example and assignments follow a common pattern, consistent with both the ideas of loop patterns [16] and concurrency patterns [17]. The pattern is designed to meet the following three goals:

1. Allow the loop that implements the single-threaded version of the problem to be reused, without any modification, in the multi-threaded version.
2. Allow the test-aware code in the single-threaded version to be reused in the multi-threaded version without requiring any additional test-aware code.
3. Demonstrate in the worked example a general pattern for implementing the fork-join algorithm that can be

implicitly followed in the assignments to make them both easy to implement and modular.

The key component of the pattern has to do with the iteration steps. It is implemented as a standalone method, called the *iterating method*, illustrated in Fig 1. The method takes as parameters a description of the portion of the problem the loop is expected to handle. In the odd and prime number problems, this description consists of an array of numbers and the start and stop index indices of the elements of the array to be handled by the loop. In the PI problem, this description is the number of iterations the loop is expected to use to do a Monte Carlo estimation of PI. In the serial version of the problem, the method is invoked by the dispatcher thread and is given the complete problem. In the concurrent version, the method is invoked by each worker thread, and is given the portion of the problem assigned to the thread.

```
public static void fillOddNumbers(int[] aNumbers,
                int aStartIndex, int aStopIndex) {
  int aNumberOfOddNumbers = 0;
  for (int index = aStartIndex; index < aStopIndex;
        index++) {
  printProperty("Index", index);
  int aNumber = aNumbers[index];
  printProperty("Number", aNumber);
  boolean isOdd = isOddNumber(aNumber);
  printProperty("Is Odd", isOdd);
  if (isOdd) {
        addOddNumber(aNumber);
        aNumberOfOddNumbers++;    }
}
```

Fig 1. Example of Iterating Method with Iterating Pattern

The loop consists of iteration, input, and output steps, which include printing of test-aware information through a predefined `printProperty` method shown in the figure. The *iteration step* gives information about the iteration number. In the odd and prime number problems, it is the index of the array element processed by the iteration. In the PI problem, it is the number of iterations performed so far. The *input step* accesses and prints the input data processed by the iteration. In the odd and prime number problems, the input data consists of the array element processed, and in the PI problem, it consists of two generated random doubles, in the range 0-1, representing X and Y coordinates of a random Cartesian point. The *output step* performs a problem-specific computation on the input and prints the result. In the odd and prime number problems, the computation determines if the input number is odd or prime, respectively, and in the PI problem, it determines if the input Cartesian point is in a circle of radius 1, that is, it calculates a hypotenuse from the origin to the X and Y coordinates and determines if it is less than the 1.0.

The input and output steps are used by the checks to determine if the computations are correct in each iteration. The iteration step is used to determine if work allocated to different threads is balanced.

The single and multi-threaded solutions also contain a *deposit method* to collect, in a global data structure, partial results. In the odd and prime number problems, it collects odd or prime number, and in the PI problem, it collects the number of iterations in which the random Cartesian point was in circle. This method is expected to be made **`synchronized`** in the multi-threaded solution. The commented version of this method in the worked-example is shown in Fig. 2.

The iterating method calls this method to deposit part or all of the results computed by the method. In the odd and prime number problem, it calls this method in each iteration that detects an odd or prime number (Fig. 1). In the PI problem, it calls this method after the loop to deposit the number of random points in the circle.

```
// Uncomment the following line to serialize access
// to the shared variables
// synchronized
public static void addOddNumber(int aNumber) {

// The first operation is redundant but is
// performed to increase race condition chances.
// Both actions are not thread safe
  totalNumberOddNumbers++;
  oddNumbers.add(aNumber);
}
```

Fig. 2 Commented Depsoit Method

Before executing the iterating method, the single-threaded solution is expected to perform an *input-processing step*, which retrieves and prints the input. In the odd and prime number problems, the input is the array of numbers to be checked for odd and prime numbers. In the PI problem, it is the total number of iterations to be used to determine PI. The input processing step becomes a *pre-fork step* in the multi-threaded solution. After executing the iterating method, the single-threaded solution performs an *output-processing step*, which retrieves the deposited data and computes and prints the final output. This step becomes the *post-join st*ep in the multi-threaded solution.

This pattern meets all of our requirements. The multi-threaded solution can completely use the iterating method, deposit method, and input and output processing steps, which also implies it does not contain any test-aware code. Thus, converting the solution to a multi-threaded one involves only steps having to do with concurrency, which include:

1. Adding the `synchronized` keyword to the header of the deposit method (Fig 2.).

2. Declaring a runnable class with the following properties. It has a constructor that takes the same arguments as the iterating method and stores these values in its instance variables. The `run` method of the class calls the iterating method with the values of these instance variables. The version of this class in the worked odd number problem is shown in Fig, 3.

3. In the code of the dispatching thread: (a) removing the call to the iterating method, (b) after the input processing step, instantiating the runnable class once for each thread to be created, passing appropriate parameters to the runnable constructor to balance work, (c) using each runnable instance to create and start a worker thread, and (d) joining the started threads after forking them and before performing the output processing step.

```
class OddNumbersWorker implements Runnable {
  int[] numbers;
  int startIndex, stopIndex;
  public OddNumbersWorker(int[] aNumbers,
         int aStartIndex, int aStopIndex) {
    numbers = aNumbers;
    startIndex = aStartIndex;
    stopIndex = aStopIndex;
   }
   public void run() {
      ConcurrentOddNumbers.fillOddNumbers(
                numbers, startIndex,
          stopIndex);
    }
}
```

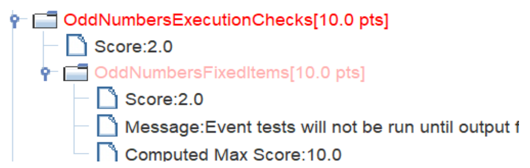
Fig. 3. Odd Number Worker (Comments Removed)

This pattern is not explicitly spelled out in the guide – participants are expected to implicitly follow it, and perhaps even explicitly derive it, based on the commented concurrent worked example.

C. *Runtime Checks*

The guide is centered around checks. Checks are executed by running a special program in the scaffolding code, which creates the GUI shown in Fig, 4 and Fig, 5, which lists the names of all checks. Each execution of the program starts a new test session. Double clicking on a check name executes the check. Unexecuted, passed, partially passed, and failed checks are colored grey, green, orange and red, respectively. Each execution of this program starts a new *test session*.

The first instruction in the guide is to execute the single runtime check for the worked example - `OddNumberFixedItems`. The correctness component of the check succeeds as it is a correct single-threaded implementation of the problem. The concurrency component of the check however fails as it is not a complete multi-threaded implementation. This action serves two purposes. First, it logs the start of the programmers' work so we can estimate how long they took. Second, it gives the programmers an idea of the steps that have been done for them in the given code and the tasks that remain.


OddNumbersExecutionChecks[10.0 pts]
Score:2.0
OddNumbersFixedItems[10.0 pts]
Score:2.0
Message:Event tests will not be run until output f
Computed Max Score:10.0

Fig, 4 Failure of Runtime Check On Given Worked Example

In the GUI, the check name is colored orange (Fig. 4) to indicate partial success. The console output, partially produced below, gives detailed information about the check status

```
Pre fork output correct
Post fork output did not match:[.*Thread.*-
>Index:.*\d.*.*, .*Thread.*->Number:.*\d.*.*, … ..
Post join output correct%0.2
```

It indicates that the pre-fork and post-join outputs are correct. The pre-fork output in our worked example is:

```
Thread 1->Random Numbers:[373, 790, 378, 226, 285, 577,
712, 411, 608, 773, 129, 189, 55, 510, 316, 530, 708, 853,
904, 567, 75, 82, 729, 115, 784, 772, 46]
```

The post-join output is:

```
Thread 1->Odd Numbers:[373, 285, 577, 411, 773, 129, 189,
55, 853, 567, 75, 729, 115]
```

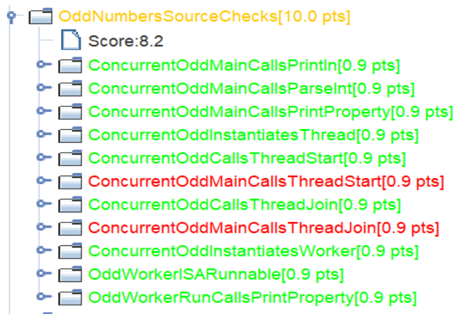
As we see, the relationship between the two is correct, with all of the input odd numbers being found. The check indicates that testing-aware output produced between pre-fork and post-join:

```
Thread 1->Index:0
Thread 1->Number:373
Thread 1->Is Odd:true
Thread 1->Index:1
Thread 1->Number:790
Thread 1->Is Odd:false
```

does not match certain expected regular expressions. The guide explains that a symptom of the problem is that all of the intermediate per-iteration output is produced by a single thread. Our runtime checks assume multiple threads must be created by the tested program. Our assignments and worked example were required to create exactly 4 threads

D. *Source Checks*

At this point, the programmers are introduced to the source checks (Fig. 5). Often students given a programming assignment say they do not know where to start or ask an instructor if they are on the right path. The source checks give them an idea of the code that must be included in a correct solution. Understanding source checks is made an optional step for those who need help with understanding the steps. The names of the tests are mostly self-explanatory. They indicate source-code milestones that must be met to solve the problem.


OddNumbersSourceChecks[10.0 pts]
Score:8.2
ConcurrentOddMainCallsPrintln[0.9 pts]
ConcurrentOddMainCallsParseInt[0.9 pts]
ConcurrentOddMainCallsPrintProperty[0.9 pts]
ConcurrentOddInstantiatesThread[0.9 pts]
ConcurrentOddCallsThreadStart[0.9 pts]
ConcurrentOddMainCallsThreadStart[0.9 pts]
ConcurrentOddCallsThreadJoin[0.9 pts]
ConcurrentOddMainCallsThreadJoin[0.9 pts]
ConcurrentOddInstantiatesWorker[0.9 pts]
OddWorkerISARunnable[0.9 pts]
OddWorkerRunCallsPrintProperty[0.9 pts]

Fig, 5 Source Check Results on Given Worked Example

As we see from the names, some of these checks test if certain classes are instantiated. An example is `ConcurrentOddInstantiatesWorker` which ensures that the runnable worker class (which has a prescribed name) is instantiated. Most of the checks test if calls to certain methods are included in the code. They provide *class-level* and *method-level source granularity* based on whether they specify the class or method that makes the call. A method makes a call if it calls the method, directly or indirectly, through a sequence of methods. A class "makes" a call if *some method* in it makes the call directly or indirectly. `ConcurrentOddCallsThreadStart` and `ConcurrentOddMainCallsThreadStart` are examples of checks that provide class and method-level granularities, respectively, determining if the class implementing the odd number problem and the main method in this class, respectively, make a call to start a thread. As we see in Fig. 5, the former succeeds while the latter fails on the given code for odd numbers. The class given to participants has a method - `createAndStartThreads` (Fig. 7)- that makes this call, but the main class runs sequential code and thus does not currently invoke this method. The distinction between these two kinds of granularities is crucial to our approach of having trainees uncomment and comment calls to understand their behavior.

The source checks verify three kinds of constraints: single-thread, multi-thread, and test-awareness. An example of single-thread constraints is the call to `parseInt`, which arises in all of our three problems, as they must convert String arguments to numbers (such as the number of random numbers to be generated). These constraints, in general, are problem-dependent. An example of a multi-thread constraint is the call to `Thread start`. Such a constraint is applicable to any fork-join problem implemented using Java fork-join abstractions. An example of a test-awareness constraint is the call to the `printProperty` method provided by our test library. Such a constraint is applicable to any application that uses our fork-join testing infrastructure.

E. *Concepts Exposed Through Tests*

The Java fork-join abstractions require, of course, an explanation of how to start and join threads. Trainees also require an explanation of synchronization of concurrent/interleaved threads. These concepts are unraveled in our model by the programmers performing guided changes to the given code.

The given code contains two methods, `concurrentFillOddNumbers and serialFillOddNumbers`, with the given main calling the latter. The first guided change is to call the former instead:

```
//serialFillOddNumbers(); // comment this line to
turn off serial processing
concurrentFillOddNumbers(); // uncomment this line to
turn on parallel processing
```

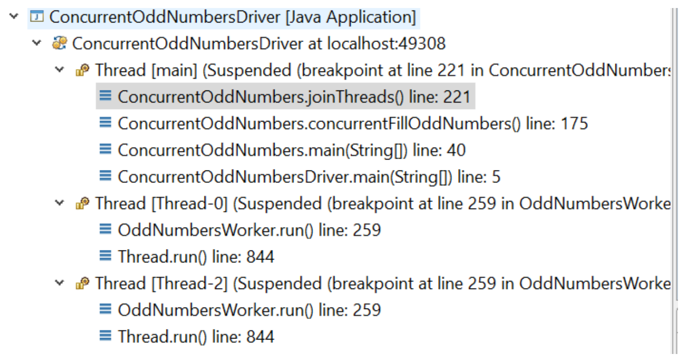
The two failed source checks in Fig, 5 now succeed. The runtime check gives more but not complete success:

```
Post fork output correct
Post join output correct
Number of forked threads correct
Pre fork events correct
Correct number of iterations
No interleaving during fork
Fork correct
Post join events correct
```

The guide explains the concepts behind the partial but not complete success: thread creation and joining. These are described in terms of the associated source checks and runtime stacks.

For example, the guide contains the following explanation of the `ConcurrentOddInstantiatesWorker` source check:

Creating a thread involves simply instantiating the predefined Thread class. Each thread is the asynchronous execution of a procedure. The term asynchronous means that the procedure that started a thread does not immediately block waiting for the thread to complete. It can do other tasks such as creating and starting other threads or not join the started thread. This means each thread is a new stack with its own base procedure at the start of the stack. The procedure that created the thread is not in this stack. If it was, it would have to immediately wait for the thread to finish. This is shown in Fig. 6 in the stack traces produced when the concurrent version of the worked example is run.



```
ConcurrentOddNumbersDriver [Java Application]
  ConcurrentOddNumbersDriver at localhost:49308
    Thread [main] (Suspended (breakpoint at line 221 in ConcurrentOddNumbers
      ConcurrentOddNumbers.joinThreads() line: 221
      ConcurrentOddNumbers.concurrentFillOddNumbers() line: 175
      ConcurrentOddNumbers.main(String[]) line: 40
      ConcurrentOddNumbersDriver.main(String[]) line: 5
    Thread [Thread-0] (Suspended (breakpoint at line 259 in OddNumbersWorke
      OddNumbersWorker.run() line: 259
      Thread.run() line: 844
    Thread [Thread-2] (Suspended (breakpoint at line 259 in OddNumbersWorke
      OddNumbersWorker.run() line: 259
      Thread.run() line: 844
```

Fig, 6 Underyling Programming System Can Aid Understanding

The base of the top stack is the main method of the main class. This stack represents a thread the underlying system automatically created when `main` was run. The four other stacks represent four threads spawned by the main thread, At the base of each stack is the predefined run method of the four instances of the predefined Thread class. This run method calls a run method specific to our problem - the run method of OddNmbersWorker. The Thread run method knows about the problem-specific method because when the Thread instance was created, its constructor was passed an instance of a class – OddNumbersWorker - that implements the problem-specific run method. We refer to such a class as a worker class, as it does the real problem-specific work. This check ensures that the main class instantiates the worker class.

F. *Leveraging the Underlying Programming Environment*

Our trainees used the command line to do their work. Had they used a visual programming environment, they could have set breakpoints to themselves create the stacks shown in Fig. 6, thereby leveraging the features provided by the underlying system to augment their learning.

An example of such use is in our explanation of interleaving. The guide explains that the tests currently show partial success because the given worked example has a delay between thread starts (Fig. 7), allowing each thread to finish before the next thread is started. Commenting out the delay gets rid of this problem, but potentially creates race conditions (exposed by the checks) because of shared access to two variables. This problem, in turn, is solved by uncommenting the synchronized keyword from the declaration of the deposit method (Fig. 2.)

These are the kind of explanations that, in the traditional model, would be provided in a lecture by a professor before hands-on programming. Here, the document and the underlying system provide this information in context, while programming and testing the example, making our model instructor-light.

V. Data

We have two main sources of data about the trainee activities: Logging of test runs and Zoom recording of the entire session.

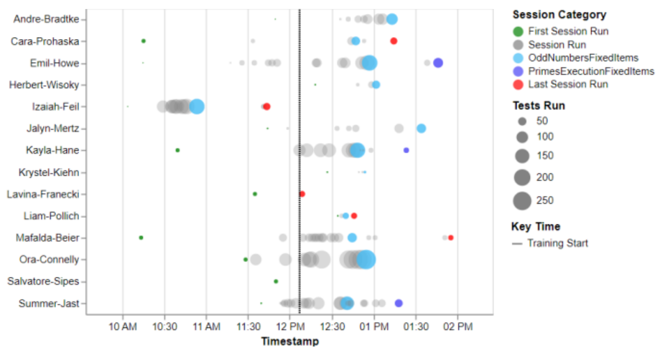
```
private static void createAndStartThreads() {
  for (int index = 0; index < workers.length;
index++) {
  threads[index] = new
        Thread(workers[index]);
  threads[index].start();
// comment out this try catch block to ensure
// concurrent_rather than serial execution of the
// thread run methods
  try {
        Thread.sleep(100);
  } catch (InterruptedException e) {
        e.printStackTrace();
  }
.
// Both actions are not thread safe
  totalNumberOddNumbers++;
  oddNumbers.add(aNumber);
}
```

Fig, 7 Leveraging the System to Explore Interleaving

A. *Logged Data*

Each test execution results in an entry with a course ID and an anonymous user ID being sent to the IBM cloud. A client de-multiplexer pulls all entries with a given course ID and distributes them in files based on the user ID, and converts the user ID into a synthesized name such as Zora West. Log analyzer and visualization programs then process these files. We have used the above architecture to create several visualizations of the activities of the participants. These were created after the workshop as we currently do not have tools to show them during a session.

Fig, 8 shows one such visualization of their progress. The X-axis shows time (EST) and the Y axis represents participants with their fake names. For all participants, we show when they executed the first (green circle) and last check (red circle) and when they completely passed the runtime check for the odd number problem (blue) and the prime number problem (purple). The size of each milestone circle is proportional to the number of tests they ran to complete the milestone. The grey circles represent test execution sessions. The vertical line marks the start of the training session (and the end of the motivational talk).



Fig, 8 Log-Based Awareness of Test Use and Milestones

These data help identify several kinds of trainees who did not follow the norm. These include those:(a) such as Jalyn, Lavina, and Ora who did not finish the odd number problem; (b) such as Emil, Summer, and Kayla who finished the prime numbers problem; (c) such as Mafalda who ran tests after finishing the odd number problem, but never finished the prime number problem presumably because of lack of time; (d) such as Izaiaiah, Mafalda and Emil, who started working asynchronously on the hands-on programming exercise before the start of the training session; (e) such as Jalyn and Lavina who ran a small number of tests; (f) such as Emil and Ora, who ran a large number of tests – presumably the optional source checks.

Together these data suggest that Jalyn and Lavina did not finish because of lack of effort, assuming they did not face hurdles that prevented them from running tests; and Ora did not finish despite test-based effort, and Emil achieved exceptional success perhaps because of an early asynchronous start.

If such information is available while the exercise is being executed, then it is possible to offer help to people like Ora who are running tests but not succeeding, or talk to people like Jalyn and Lavina, who are not running any tests to find the reason. This is important for students too shy to ask for help [12-14];

If such information is available after the exercise, then it is possible to identify how much time it takes different kinds of trainees to complete milestones. In a traditional course, it can be used to determine if the number of hours students are putting in is near the expected number based on the course credits.

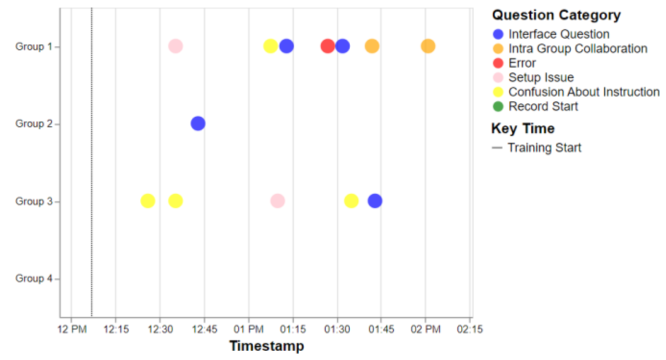
Together, these data suggest that most trainees who did not finish the second milestone stopped testing much before the session end time. On the other hand, it also shows some who tested after the first milestone such as Cara and Mafalda did not complete the second milestone, implying that the time devoted to the training session (around 100 minutes) was not sufficient for some participants to complete the prime number exercise. No one started the optional PI assignment. In retrospect, this is not surprising as the guide was 26 pages long (with figures) and involved understanding and experimenting with dense commented Java code. These data also indicate that the three trainees who finished the prime number problem took much less time and ran fewer tests to reach this milestone than the previous milestone of following the guide to solve the odd number worked-example, showing the effectiveness of the instructor-less guide as a teaching tool. Finally, the data suggest the possibility of doing asynchronous work, before the training session, which may have been the reason for Emil's success.

As we see above, this visualization is designed for a test-based synchronous short-duration session in which trainees do most of their work in that session and possibly a little before and after the session. It complements our previous research on visualizations describing long-duration (multiple days and weeks) asynchronous work on concurrency assignments carried out in homework assignments using logging of not only tests but also programming environment commands [18].

B. *Zoom Recording*

The Zoom recording was used to identify the usefulness of a synchronous session in which graduate assistants helped overcome hurdles faced by the trainees working in groups. It was used to identify (a) times when questions were asked both within the group and to the assistants, and (b) the nature of the question. Not all questions were identified because the camera captured only one group and the audio was not picked up uniformly.

Fig. 9 visualizes this information. Again, the X-axis shows absolute times (EST) and the vertical line marks the start of the training session. The Y-axis now shows groups rather than individual users. The questions were classified into five categories, each of which is associated with a different color: (a) Questions about interfaces (purple); (b) questions answered by a group member through intra-group collaboration - the nature of the question could not be deciphered (beige); (c) confusion about an error message (red); and (d) set up issues (pink).



Fig, 9 Zoom-based Awareness of Problems and Help Seeking Behavior

Even though we did not capture all questions, the data show the usefulness of graduate assistants providing help throughout the session. The wide prevalence of (purple) interface questions suggests one such reason – this is a Java concept missing in C++ - and it is likely such concepts were unfamiliar to many participants. The wide prevalence of (yellow) instruction questions shows that our first attempt at writing an instructor-less guide has room for improvement. The presence of red dots shows that the error handling in our checks needs improvement. The absence of any questions from group 4 – consisting of a remote participant - suggests the importance of in-person help.

VI. Conclusions

This paper makes several contributions. It is the first to show the need for studying the training of trainers as a first-class issue separate from training students. Some of the differences our data show are the uneven background - some participants needed help with interfaces – and motivation - two trainees apparently did little work to meet the first milestone, and most stopped after the first one, despite having time. It is doubtful students being graded for such work would have shown such behavior. These findings are enabled by our automatic logging of tests and manual analysis of Zoom recordings – mechanisms unique to our work - and novel visualizations of these data.

The pedagogical model with a focus on multi-threading, a pattern for converting single-thread code to multi-thread fork-join code, concepts descriptions centered around checks, and in-context exposition of concepts through experimentation enabled by the underlying programming environment and checks is the most important conceptual contribution. It is built on a new model of fork-join algorithms and Java fork-join abstractions.

Further work is of course needed to carry out a follow-up larger experiment – involving both faculty and students - and porting of our checks to other concurrency models and abstractions,

References

[1] Ko, Y., B. Burgstaller, and B. Scholz, Parallel from the beginning: the case for multicore programming in thecomputer science undergraduate curriculum, in Proceeding of the 44th ACM technical symposium on Computer science education. 2013, ACM: Denver, Colorado, USA. p. 415-420.

[2] Adams, J.C., Injecting parallel computing into CS2, in Proceedings of the 45th ACM technical symposium on Computer science education. 2014, ACM: Atlanta, Georgia, USA. p. 277-282.

[3] Geist, R., J.A. Levine, and J. Westall, A problem-based learning approach to GPU computing, in Proceedings of the Workshop on Education for High-Performance Computing. 2015, ACM: Austin, Texas. p. 1-8.

[4] Lupo, C., Z.J. Wood, and C. Victorino, Cross teaching parallelism and ray tracing: a project-based approach to teaching applied parallel computing, in Proceedings of the 43rd ACM technical symposium on Computer Science Education. 2012, ACM: Raleigh, North Carolina, USA. p. 523-528.

[5] Burtscher, M., W. Peng, A. Qasem, H. Shi, D. Tamir, and H. Thiry, A Module-based Approach to Adopting the 2013 ACM Curricular Recommendations on Parallel Computing, in Proceedings of the 46th ACM Technical Symposium on Computer Science Education. 2015, ACM: Kansas City, Missouri, USA. p. 36-41.

[6] Graham, J.R., Integrating parallel programming techniques into traditional computer science curricula. SIGCSE Bull., 2007. 39(4): p. 75-78.

[7] Neelima, B. and J. Li, Introducing high performance computing concepts into engineering undergraduate curriculum: a success story, in Proceedings of the Workshop on Education for High-Performance Computing. 2015, ACM: Austin, Texas. p. 1-8.

[8] Brown, R. and E. Shoop, CSinParallel and Synergy for Rapid Incremental Addition of PDC Into CS Curricula, in Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum. 2012, IEEE Computer Society. p. 1329-1334.

[9] Brown, R. and E. Shoop, Modules in community: injecting more parallelism into computer science curricula, in Proceedings of the 42nd ACM technical symposium on Computer science education. 2011, ACM: Dallas, TX, USA. p. 447-452.

[10] 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.

[11] 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.

[12] Begel, A. and B. Simon. Novice software developers, all over again. in International Computing Education Research Workshop. 2008.

[13] LaToza, T.D., G. Venolia, and R. Deline. Maintaining mental models: a study of developer work habits. in Proc. ICSE. 2006. IEEE.

[14] Allison, M.R., P.R. Pintrich, and C. Midgley, Avoiding Seeking Help in the Classroom: Who and Why? . Educational Psychology Review, 2001. 13(2).

[15] Dewan, P., A. Wortas, Z. Liu, S. George, B. Gu, and H. Wang. Automating Testing of Visual Observed Concurrency. in 2021 IEEE/ACM Ninth Workshop on Education for High Performance Computing (EduHPC). 2021. IEEE.

[16] Astrachan, O. and E. Wallingford. Loop Patterns. in PLoP. 1998.

[17] Adams, J., R. Brown, and E. Shoop, Patterns and Exemplars: Compelling Strategies for Teaching Parallel and Distributed Computing to CS Undergraduates, in Proceedings of Parallel and Distributed Processing Workshops and PhD Forum. 2013, IEEE Computer Society. p. 1244-1251.

[18] Dewan, P., S. George, B. Gu, Z. Liu, H. Wang, and A. Wortas. Broad Awareness of Unseen Work on a Concurrency-Based Assignment. in 2021 IEEE 28th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW). 2021. IEEE.