Parallel X: Redesigning of a Parallel Programming Educational Game with Semantic Foundations and Transfer Learning

Devon McKee
dlmckee@ucsc.edu
University of California Santa Cruz
Santa Cruz, California, USA

Zhiyu Lin
zlin34@ucsc.edu
University of California Santa Cruz
Santa Cruz, California, USA

Boyd Fox
thomas.boyd.fox@gmail.com
Independent
Williamsburg, Virginia, USA

Jiahong Li
jli906@ucsc.edu
University of California Santa Cruz
Santa Cruz, California, USA

Jichen Zhu
jicz@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Magy Seif El-Nasr
mseifeln@ucsc.edu
University of California Santa Cruz
Santa Cruz, California, USA

Tyler Sorensen
tsorensen@microsoft.com
Microsoft Research
Redmond, Washington, USA

Abstract

The vast computational power enabled by parallel programming has led to major advances in data science, scientific computing, and AI. Yet, teaching parallel programming remains challenging: non-deterministic execution introduces an exponential number of possible execution paths, where a single flawed path can compromise an entire program. Games have shown promise in helping students grasp complex concepts, but prior game-based approaches to teaching parallelism have struggled with two key limitations: a lack of strong semantic grounding in concurrency concepts, and weak transfer from visual gameplay to real-world coding practices.

In this paper, we present *Parallel X*, a redesigned educational game that directly addresses these limitations. It (1) incorporates classic concurrency theory to support debugging tasks, an increasingly vital skill as AI-generated code becomes more common and error-prone, and (2) introduces transfer learning activities that connect in-game states to C++ code, bridging the gap between conceptual understanding and implementation. A usability study with undergraduate CS majors shows improved usability scores and significantly higher ratings for information accessibility compared to an earlier version of the game.

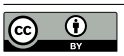
CCS Concepts

• **Applied computing** → **Computer-assisted instruction**.

Keywords

Concurrency, Learning transfer, Debugging, Educational tooling

ACM Reference Format:
Devon McKee, Zhiyu Lin, Boyd Fox, Jiahong Li, Jichen Zhu, Magy Seif El-Nasr, and Tyler Sorensen. 2026. Parallel X: Redesigning of a Parallel Programming Educational Game with Semantic Foundations and Transfer Learning. In *Proceedings of the 57th ACM Technical Symposium on Computer Science Education V.1 (SIGCSE TS 2026), February 18–21, 2026, St. Louis, MO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770762.3772657



SIGCSE TS 2026, St. Louis, MO, USA

ACM ISBN 979-8-4007-2256-1/2026/02
https://doi.org/10.1145/3770762.3772657

1 Introduction

Parallel programming is central to important areas in computer science, e.g., data science and machine learning, but it remains notoriously difficult to master. This difficulty arises from rare bugs caused by non-deterministic execution. These challenges extend to the classroom, where educators have explored innovative approaches to make parallelism more accessible. In particular, educational games have shown promise in this area.

For example, Parallel [30] is a game that encodes parallel programming concepts as puzzles and players solve them by avoiding potential errors such as data races and deadlocks. *Parallel Islands* [2] is another educational game that consists of classroom activities where students act as agents in a concurrent system. *BlocklyPar* [37] is a modified version of Google's visual programming editor and focuses on introducing students to concurrency concepts like parallel speedups using block-based programming puzzles.

Despite innovations in this space, prior educational games for parallel programming have had shortcomings that limit their effectiveness in college classrooms, where the emphasis is often on foundational semantics and practical coding skills rather than broad conceptual overviews. In particular, existing tools have two gaps:

- **Semantic grounding**: The core semantics of parallel execution involve interleaved sequences of atomic actions performed by each thread. However, existing games tend to not have a structured way to pause and resume execution which obscures this critical model and weakens the connection to how real parallel programs behave.
- **Transfer learning**: While visual metaphors can make games engaging, they can diverge from real-world applications. This is especially evident in the handling of shared resources, which

may appear visually unified but in actual code they are scattered and indirect (e.g., distinct pointer variables across functions).

Parallel X. In this work, we present *Parallel X*, an educational game designed to incorporate foundational parallel programming semantics and support explicit transfer learning. *Parallel X* features a redesigned game engine that addresses the shortcomings identified above. Specifically, it introduces two new components:

- **Formal semantic engine (Sec. 3.1)**: Core game elements, such as threads, semaphores, and packages, are formalized in TLA+ [16], a well-established specification language based on the interleaving of atomic actions. The game state can be mapped to a corresponding TLA+ model, enabling formal validation and reinforcing the underlying concurrency semantics.
- **Game-to-code visualizations (Sec. 3.3)**: Beyond formal modeling, the game also generates corresponding C++ code for each game state. A dynamic mapping highlights lines of C++ code in real time as game threads execute, helping students connect in-game actions to actual parallel constructs.

These new components enable new educational features that support a deeper understanding of parallel programming, namely:

- **Explicit debugging (Sec. 3.4)**: Students can now single-step threads, with each step representing a single atomic action, as defined by the formal model. This mirrors the behavior of real-world debugging tools for parallel programs (e.g., GDB [38]). Furthermore, teaching students how to diagnose and debug programs (parallel or not) is arguably as important as teaching code development itself [41].
- **Interactive tutorials (Sec. 4.2)**: Earlier work presented concurrency errors, e.g., deadlocks and data races, with static explanations and animations [30]. With the new debugging tools, tutorials now ask students to actively reproduce these errors by guiding threads into faulty states, closely mimicking how such bugs are identified in practice.
- **Learning transfer (Sec. 4.3)**: With game-to-code visualizations, students can see how in-game actions map directly to parallel C++ code. As threads execute atomic actions in the game, corresponding lines of code are highlighted in real time, promoting transfer of learning to real-world programming tasks.

To establish *Parallel X*'s utility as an effective educational tool, we perform a usability study with eight computer science undergraduate students and a series of heuristic evaluation tests, discussed in Sec. 5.

2 Related Work

There is a rich history of educational games for computer science education, especially around programming. These games provide diverse approaches ranging from block-placement interfaces [32] that emulate concrete programming constructs, to puzzle and simulation games [11, 15] that emphasize task-oriented learning with a game-first design. Some programming education games even contain competition and continuous improvement components [12]. The integration of games and game mechanics for STEM education and learning have demonstrable advantages, including enhanced self-confidence in programming abilities [21], increased motivation for learning, and a correlation to improved academic outcomes [42].

As discussed in Sec. 1, several educational games have been developed to teach parallel programming [2, 3, 30, 37]. Other tools have been created to aid student learning by simulating concurrent programs with visualizations of debugging activities [22, 39]. Within this domain, *Parallel* emerges as a notable example for its strong visual metaphor and robust tooling for visualizing user data that has been discussed in several publications [23, 44, 45], and as such, it serves as a strong foundation for our work.

Parallel consists of several arrows (threads) that move along a track (a program) in a non-deterministic way. Their goal is to deliver packages (shared memory). However, without sufficient coordination (synchronization), the arrows can collide or block each other in erroneous ways (data races and deadlocks). Thus, the user needs to judiciously place semaphores (both acquires and releases) [13, Ch. 8] around the track, relating to how a programmer would add the same components to their parallel code.

While educational games such as *Parallel* have shown promise, they must carefully consider their design to effectively support knowledge transfer [25]. In some cases, it has been shown that game objectives are prioritized over educational goals, limiting deep learning and failing to engage all learners [21]. Even worse, reliance on game elements can undermine long-term learning by encouraging superficial interaction rather than meaningful understanding [6]. Further, while it is common for learning games to have a theory of transformation [35], most work on educational games for parallel programming do not consider this in their learning plans.

3 *Parallel X* Design

In this work, we aim to address some of the potential pitfalls of educational games by introducing *Parallel X*, a new game built on the foundations of *Parallel*. *Parallel X* places a deeper emphasis on parallel programming semantics to support conceptual accuracy, improve tutorials, and promote the transfer of learning to real-world code. It features a rigorous semantic design intended to enhance knowledge transfer by aligning gameplay with core concurrency concepts and a structured lesson plan grounded in a theory of learning transformation. More information on the game design of Parallel and its community-based learning visualization, which the following features are built on, can be found in [24, 43].

We introduce a formal specification for the game (Sec. 3.1) and demonstrate its use in supporting tools and educational material, including a formal model checker (Sec. 3.2), a game-to-code engine (Sec. 3.3), and a redesigned lesson plan (Sec. 4.2).

3.1 A Semantic Foundation

First, we establish the core semantic foundation of *Parallel X*. This description aims to be as formal as possible, and thus can be straightforwardly implemented in a logic engine (e.g., model checker), which we do in TLA+ (Sec. 3.2).

Our formalism begins with the game *objects*. The first being a *location*, which corresponds to a program instruction. A location may contain a *component*, e.g., semaphores and pickups, each of which have their code counterparts (Sec. 3.3). A *track* is a graph of locations, and represents a program. A *thread* is an object that

has (1) a location on the track; (2) a path to follow on the track; and (3) an optional package to deliver.

A *game state* consists of the game objects. A game state has a single track and initial thread locations. There are various components placed throughout the track, and the user can add additional components (e.g., semaphores) to constrain the execution. Following classic semantic modeling techniques, the rules for how the game state evolves are given as a series of *transitions*. A transition is given by a condition (guard), and then an action which updates the state. If the guard is satisfied, then it is possible for the transition to take place.

The transition rules state that each thread is allowed to move to the next location along its track. Given this, there can be multiple choices for the next transition; and an execution is allowed to arbitrarily pick one. A complete execution, i.e., a series of transitions until competition, is called a *schedule*. In a correct parallel program, there must not be errors (formalized below) for any possible schedule. To guard against erroneous schedules, certain components can act as a guard (i.e., block) a thread transition.

Components. We now list the game components we model, along with their impact on the thread transitions and game state. We also describe their real-world code counterparts:

- **Semaphore/Signal**: Semaphores can be either *open* or *closed* (with a configurable initial state). If a thread is in a location immediately prior to a semaphore and the semaphore is closed, then the thread cannot progress. Semaphores are connected to a signal. If a thread lands on the location with a signal, then the state of the semaphore is toggled. Semaphores are analogous to `acquire()` calls on real-world semaphores [4], while signals are analogous to `release()` calls.
- **Pickup/Delivery**: Allows threads to pick up and deliver packages. Various types of pickups/deliveries exist, which cause threads to pick up different package types — pickup and delivery are analogous to shared memory accesses.
- **Exchange**: Exists as a pair: if a thread is on one exchange location, then it is blocked until another thread ends up on the paired exchange location. When both threads are on an exchange, they swap packages and continue. This construct is analogous to thread-to-thread communication.
- **Diverter**: Directs threads in different directions on the track depending on the package type they are carrying, analogous to control flow based on thread-local values.
- **Direction Switch**: Has an internal state that can direct threads in a variety of directions along a track. The internal state can be toggled by a signal and is analogous to control flow switching on shared memory variables.

Errors and Goals. This leads us to the possible errant states for the game, which include deadlocks, data races, and goal failures. Deadlocks are a state in which there are no possible subsequent states, i.e., all thread transitions are blocked, which happens when all threads are either waiting at a closed semaphore or an exchange point. A data race is a state in which two threads are occupying the same location at a delivery while carrying a package — this is a condition that has possible future states, but is considered a failure due to it leading to undefined results for shared memory accesses in real-world code [1]. Finally, a goal failure state is a state in which too many packages have been delivered by a single thread, or to a single delivery.

3.2 Model Checking Solutions

Determining whether a parallel system is correct is difficult due to the exponential number of possible schedules. Even if a system behaves correctly under most schedules, a single errant schedule can invalidate its correctness. Other parallel games typically rely on random testing, where threads are randomly executed over many iterations. However, testing parallel systems this way has well-documented limitations, e.g., see [27].

However, the semantic foundation of *Parallel X*, as described in Sec. 3.1, is natural to implement in a formal modeling tool, such as TLA+ [17]. TLA+ is a framework (a language and a model checker) that can efficiently check all possible schedules for parallel systems; and in fact, it has been used to verify many complex real-world parallel and distributed systems [14, 28, 31].

To take advantage of this rigorous formal tooling, we create a translation mapping from *Parallel X* game states to PlusCal [19], a higher-level front-end language for TLA+. Once a student submits the solution the game state is sent to the translation tool, which translates the game into a grid-based intermediate representation. We map this representation to a PlusCal specification that (1) checks for errant game states and (2) implements thread transition rules for game components. The PlusCal specification is then translated into the lower-level TLA+ language and fed to TLC [18], a parallelized model checker that explores schedules in parallel and searches for errors. If an error is found, the tooling returns which condition failed and the sequence of transitions that caused the failure.

As expected, we find that our submission backend is effective at detecting errant states in all student solutions within a reasonable amount of time (approx. 3 – 60 seconds), while also exploring a massive number of game states (approx. 1K – 100K), thanks to a mature and well-developed model checking engine. When a failure is detected, the backend provides students with the specific failure condition, along with debugging tips to help identify the underlying issue. In future work, we plan to replay the errant schedule from the TLA+ trace to further aid student understanding. In contrast, we found that the existing *Parallel* tools commonly accepted incorrect solutions when asked to "test" the submission, passing an incorrect solution approximately 33% of the time in 30 testing iterations.

By providing *Parallel X* with a formal backend built on rigorous tools used in real-world parallel system verification, we ensure that the game's semantic foundation aligns with authentic parallel programming concepts. Beyond supporting gameplay, this formalism also holds potential as an educational tool in formal verification courses, where tools like TLA+ are commonly taught. In fact, methods that teach formal methods through puzzles already exist and are well-regarded within the community, both as conference papers [34, 36] and tutorial blog posts [46].

3.3 Translating Game to Code

While formal verification tooling is effective at checking solutions, knowledge transfer in parallel programming requires code representation. Given the tooling that was created in Sec. 3.2, a natural

Table 1: *Parallel X* component to C++ code construct mapping

Game Component	Code Construct
Semaphores	Semaphore `acquire()`
Signals	Semaphore `release()` OR updates to atomic memory flags
Pickups	Data source retrieval
Deliveries	Shared variable updates
Diverters	If statements predicated on packages
Direction Switches	If statements predicated on atomic memory flags

extension is to create executable parallel code in a common parallel programming framework. In this case, we picked C++ and used its standard concurrency libraries, e.g., `thread` [5] and `semaphore` [4]. The tooling first creates a grid-based map of the game state. This map is then evaluated for each thread placed on the map, and a sequence of locations along the track are generated. The thread paths are then organized into directed graphs, which are then traversed by a code generation step. An example of these graphs is shown in Fig. 1b, which shows a thread path generated from level 5 of *Parallel* (shown in Fig. 1a). From these graphs, the code generation step creates corresponding lines of code for special components, and inserts control flow where necessary. The code generation step also exports the code with annotations to perform line-level code tracking as threads are animated throughout the game (not shown here). The resulting code is shown in Fig. 1c.

Game components map nearly one-to-one with single-line program statements. Table 1 summarizes the mapping: the components that redirect threads, i.e., diverters and direction switches, require some control flow to be inserted, but remain relatively simple. Contrary to package delivery, which is done non-atomically and thus requires proper synchronization, direction switches operate atomically as their updates must be immediately reflected between thread step executions. Exchanges similarly perform an atomic exchange of the packages between threads, as they would otherwise constitute a data race. Through this mapping, we can take arbitrarily complex student solutions and produce analogous code.

3.4 Parallel Debugging

Now that we have formalized the notion of a game state and how it is updated by threads, it is straightforward to introduce debugging functionality, e.g., similar to widely used tools like GDB [38]. Debugging is important as students are presented with more complex problems, especially in code, as thread schedules can be difficult to reason about without hands-on experience. To support debugging within *Parallel X*, we create new tooling within the game to allow students finer grained control over thread execution, reflecting the kind of tooling available in real-world code.

With the formal specification of *Parallel X*, students can view executions of the game as atomic steps for each thread that interact with the level state. We introduce single-step controls for each thread, allowing the execution schedule of the game to be controlled by the user. This allows users to recreate specific game states, including failure conditions. These single-step controls advance the animation of the game forward in each step, so users can plainly see where failure conditions such as data races, deadlocks, or incorrectly delivered packages can occur.

4 Game-Based Learning Design

We now present an expanded lesson plan for students that consists of newly designed activities and groups of levels reorganized into units, to form a complete curricula for students playing the game.

4.1 Activities

Levels in *Parallel X* consist of an initial game state and an activity type — either *Synchronization* or *Debugging*.

Synchronization Activities. In these activities, students must place synchronization components on the game track to avoid errant schedules. Students must also properly synchronize the threads such that all packages are delivered as expected. Synchronization activities require students to submit their solution to the model checker backend in order to advance, which will return a failure if any errant schedules exist in their solutions.

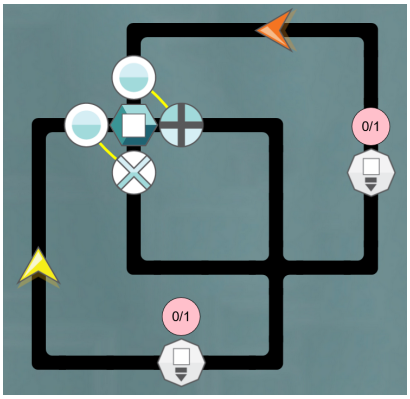
Debugging Activities. Given that students are often reluctant to use debuggers [9], we instead build debugging into the game directly. Within debugging activities, the only controls exposed to the students are the new thread stepping buttons (Sec. 3.4), which mirrors the controls in real-world parallel code debuggers [26]. The controls for adding additional synchronization components are disabled. Instead, the student is given a goal to manually guide the threads to a certain schedule. In many cases, this property is to expose an errant state, e.g., deadlock or data race. This mirrors how debugging often occurs in the real world. Given this, students should come out of these activities understanding how thread execution occurs by creating their own schedules.

4.2 Redesigned Lesson Plan

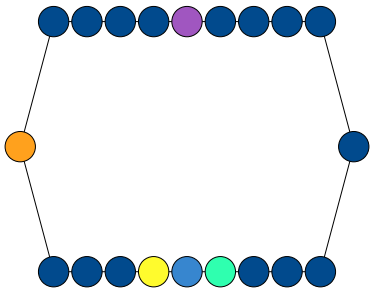
The theory of transformation within the educational game design field defines a journey of how learners' knowledge or skills evolve. As discussed in the book *Designing Serious Games*, "Building a theory of change for your game helps you break down how the player's knowledge or understanding might change over the course of experiencing your game, how their skills and/or abilities improve or develop during gameplay, or how their opinions or worldview evolve as a result of playing your game" [35]. To support student learning, an added contribution in *Parallel X* is the development of a theory of change, provided as a new lesson plan where levels are designed alongside the new activity types.

In *Parallel X*, the levels are structured in various units as part of this larger lesson plan, which focuses on introducing one concurrency concept per unit with various activities. Each unit consists of 1-3 debugging activities followed by 1-3 synchronization activities, allowing students to examine the semantics of newly introduced components before asking them to create solutions with them.

As an example of the introduction of concepts to students in this lesson plan, we examine unit 3, which concerns deadlocks in *Parallel X*. This unit contains a debugging activity to have students manually create a schedule that will reach a deadlocked state; afterwards, there is a synchronization activity that asks them to add sufficient synchronization to avoid deadlocks in a more complex example. Fig. 2 shows these activities. Fig. 2a shows level 3.1, which includes a debugging activity in which students should step the threads forward to reach a deadlock. Both threads will initially pass the



(a) Level 5 in *Parallel* with a solution created by placing two semaphores and signals to guard thread pickups and deliveries.

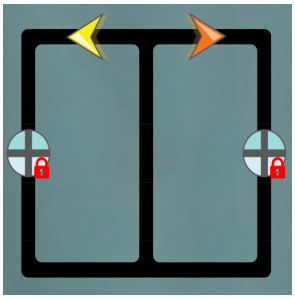


(b) A thread path generated from the game-to-code compilation tooling from *Parallel* level 5. Darker blue nodes represent empty spaces, orange nodes represent thread locations, lighter blue nodes represent pick-ups, purple nodes represent deliveries, while cyan and yellow nodes represent semaphores and signals.

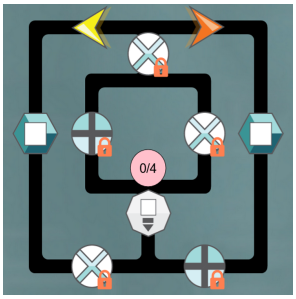
```
void thread_func0() {
    bool carrying_package = false;
    for (int i = 0; i < LOOP_MAX; i++) {
        sem1.acquire();
        carrying_package = true;
        sem0.release();
        if (carrying_package) delivery1++;
    }
}

void thread_func1() {
    bool carrying_package = false;
    for (int i = 0; i < LOOP_MAX; i++) {
        sem0.acquire();
        carrying_package = true;
        sem1.release();
        if (carrying_package) delivery0++;
    }
}
```

(c) An excerpt from the C++ thread code generated from *Parallel* level 5. Each thread is given its own thread function, which includes each statement added in the code generation step. Some preamble is generated, which is excluded here.

Figure 1: Various stages from the game-to-code pipeline.



(a) Level 3.1 in *Parallel X*, which illustrates deadlocks and requires the user to create an errant schedule.



(b) Level 3.2 in *Parallel X*, which asks students to create a solution which will not deadlock in any schedule.

Figure 2: Deadlock examples from *Parallel* and *Parallel X*.

open semaphores once, but since there are no signals to reopen them, they will get deadlocked on the second cycle around the level. This is accompanied by tutorial panels, which guide students through this understanding. Fig. 2b shows level 3.2, which includes a synchronization activity in which students are intended to place signals that will allow the threads to pass through the preplaced semaphores and deliver the packages as intended.

This design philosophy is reflected in the rest of *Parallel X*'s lesson plan, which introduces concepts in discrete units, with each concept consisting of both debugging and synchronization activities. This design encourages students to more deeply engage with the semantics components of the game.

4.3 Transferring Knowledge from Game to Code

The final unit of the redesigned lesson plan consists exclusively of transfer activities, which introduce students to how the various synchronization objects in the game appear in C++, and how error conditions like deadlocks and data races show up. Transfer activities are introduced as modified debugging activities, which include single-step thread controls and utilize the game-to-code translation tooling mentioned in Sec. 3.3 to display analogous C++ code.

5 Evaluation

To perform an initial evaluation of *Parallel X*, we conducted a pilot A/B usability study with eight undergraduate computer science participants. We select usability as an easily assessable metric for improvements in interaction with the system, but further assessment of the effects on student learning is necessary. In accordance with usability engineering work, only 3-5 participants are needed to test usability [29]. Participants were randomly assigned to condition A (*Parallel*) or condition B (*Parallel X*). Participants completed a short introduction and then played a sequence of levels. We chose specific levels from *Parallel* and *Parallel X* that capture the same parallel programming concepts. To maintain a consistent flow, each level was allocated a 10-minute time limit; and the entire session was capped at 40 minutes. Participants' actions were observed by a researcher, and their questions, feedback, or technical issues were documented. Following the gameplay, participants were administered a post-survey questionnaire, based on the Post-Study System Usability Questionnaire (PSSUQ) [20], including questions about their perceptions of the game's usability. Participants received a $25 gift card as compensation.

We conducted a comparative analysis between these two conditions. The results indicate a slight improvement in overall usability, with a mean score of 5.58 ± 1.29 for Condition B (*Parallel X*) compared to 5.46 ± 1.31 for Condition A (*Parallel*). Notably, Condition B demonstrated a 2-point increase in the response to the question "It was easy to find the information I needed," which aligns with the intended enhancements from the redesigned tutorials and solution verification process. These findings suggest promising outcomes for

the redesigned version in enhancing user experience, particularly in terms of information accessibility.

In addition to the comparative analysis, as discussed above, we collected observational records that provide further insights into the impact of the features discussed in Section 3 for *Parallel X*.

Despite explicitly instructing participants to progress through levels sequentially, one participant in Condition B adopted a strategy of revisiting previous levels when encountering difficulties to fully utilize the lesson plan. In contrast, Condition A participants reported a difficulty spike between levels, and one expressed frustration due to the inability to review previous tutorials. This challenge was not observed in *Parallel X*, as the lesson plan included debugging levels that provided hints to support completion of levels. This highlights the advantage of the new lesson plans, particularly the debugging activities which offer support to improve learning outcomes.

Participants also preferred the game content over traditional text-based tutorials. While multiple participants attempted to skip the text tutorials, a participant in Condition A struggled due to this behavior, whereas one participant in Condition B, despite also skipping tutorials, successfully became the only participant to complete the lesson plan. These individual cases suggest the potential of the redesigned lesson plan to enhance educational efficacy beyond the original *Parallel*, without conventional text-based approaches.

However, several elements in *Parallel X* introduced new challenges. Participants reported difficulties navigating the game and interacting with the redesigned features. For instance, one participant did not realize the need to place objects during the first synchronization level, as this was not required during previous debugging levels. Similarly, some participants struggled to understand how to submit a solution (after testing). While the new lesson plan provided enhanced support for achieving educational goals, further measures are needed to address the added complexity.

Limitations. This study was limited to testing the introductory portion of the lesson plan, restricting our ability to assess the usability of the game's later levels, particularly those involving game-to-code transfer. Out of all participants, only one participant successfully completed the lesson plan within the time limit, underscoring the challenges of teaching parallel programming even with the redesigned support. Future studies should consider extending the scope to explore deeper into the lesson plan and better evaluate how students transfer their learned knowledge to practical parallel programming skills beyond the game.

Heuristics. We also conducted a heuristic evaluation of the game design of *Parallel X* [7, 8]. Three researchers who are not involved in the design of Parallel X systematically applied 72 heuristics drawn from PLAY [7] and GAP [8], allowing the team to identify recurring strengths and weaknesses across the design. The results of the review showed strong potential for *Parallel X* to deliver engaging, easy-to-learn but hard-to-master gameplay, that supports a well-controlled and fatigue-free learning experience, with it meeting a majority of the heuristics, at least 70% for all reviewers. The common point to improve was the lack of: (1) clear, immediate, fun; and (2) multimodal feedback in the tutorials and during gameplay, which may hinder students' navigation through the test-and-submit flow. Accordingly, we plan targeted improvements before integrating the system into a larger study on learning and metacognitive outcomes.

6 Discussion

The semantic foundations and corresponding features in *Parallel X* provide several avenues for future work. For example, the model checking engine supports deeper analysis, such as determining the optimality of solutions (for instance, minimizing synchronization) and enabling intelligent hinting systems for incorrect solutions. It also allows for rapid validation of game states, which opens up possibilities for procedurally generating new levels. Transfer learning activities can be extended to a variety of programming languages, especially those with native semaphore support such as Rust [40], Go [10], and Ruby [33]. These activities are designed to fit naturally into classroom settings, including hands-on debugging tasks guided by an instructor.

Initial results from the usability study reinforce the potential of *Parallel X* for integration into classroom curricula. Students showed clear improvement in their ability to locate relevant information, addressing a common issue identified in earlier studies of Parallel. However, to fully understand the impact of *Parallel X* on student learning, especially in relation to the new transfer learning components, further investigation is needed. These components were not included in the current study because they are unique to *Parallel X* and could not be evaluated through A/B testing. To address this, we plan to conduct a larger study comparing learning outcomes between students using *Parallel X* and those using the original *Parallel*, leveraging built-in visualization tools to support both instruction and analysis [23, 45]. A central focus of this future work will be to assess how well students transfer in-game concepts to real-world challenges, such as debugging complex parallel code.

7 Conclusion

Our work introduces *Parallel X*, a novel educational tool that empowers students to grasp critical parallel programming concepts. The preliminary results are encouraging, showing that students readily embrace *Parallel X*, which highlights its potential to become a helpful addition to parallel programming curriculum. This work establishes a semantic foundation that future work on *Parallel* will build on, and markedly contributes to parallel programming teaching among serious games. The game is publicly available at https://code.playparallel.com for further review.

Acknowledgments

We thank the reviewers of this paper for their help and feedback in refining this paper. We also thank the members of the *Parallel* team, whose work serves as the foundation for *Parallel X*. In particular we thank Brian Smith, who provided significant input on this paper, and the team at UCF (including Roger Azevedo and Cameron Marano) whose insights into student learning informed much of the design of *Parallel X*. This material is based upon work supported by the National Science Foundation under Award No. 2302778. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] Sarita Adve. 2010. Data races are evil with no exceptions: technical perspective. *Commun. ACM* 53, 11 (Nov. 2010), 84. doi:10.1145/1839676.1839697

[2] Melissa Cameron. 2023. *Parallel Islands: A diversity aware tool for parallel computing education.* Ph. D. Dissertation. Virginia Tech.

[3] Tomorrow Corporation. 2018. *7 Billion Humans.* Tomorrow Corporation. https://tomorrowcorporation.com/7billionhumans

[4] cppreference. 2025. std counting_semaphore std binary_semaphore. https://en.cppreference.com/w/cpp/thread/counting_semaphore.html

[5] cppreference. 2025. std::thread. https://en.cppreference.com/w/cpp/thread/thread.html

[6] Luis De-Marcos, Adrián Domínguez, Joseba Saenz-de Navarrete, and Carmen Pagés. 2014. An empirical study comparing gamification and social networking on e-learning. *Computers & education* 75 (2014), 82–91.

[7] Heather Desurvire and Charlotte Wiberg. 2009. Game usability heuristics (PLAY) for evaluating and designing better games: The next iteration. In *International conference on online communities and social computing*. Springer, 557–566.

[8] Heather Desurvire and Charlotte Wiberg. 2015. User experience design for inexperienced gamers: GAP—Game Approachability Principles. In *Game user experience evaluation*. Springer, 169–186.

[9] Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. 2010. Debugging From the Student Perspective. *IEEE Transactions on Education* 53, 3 (2010), 390–396. doi:10.1109/TE.2009.2025266

[10] Google. 2025. Semaphore package - Go Packages. https://pkg.go.dev/golang.org/x/sync/semaphore

[11] Lindsey Ann Gouws, Karen Bradshaw, and Peter Wentworth. 2013. Computational thinking in educational activities: an evaluation of the educational game light-bot. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 10–15.

[12] Ken Hartness. 2004. Robocode: using games to teach artificial intelligence. *Journal of Computing Sciences in Colleges* 19, 4 (2004), 287–291.

[13] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming* (first ed.). Morgan Kaufmann.

[14] Young-Mi Kim and Miyoung Kang. 2020. Formal Verification of SDN-Based Firewalls by Using TLA+. *IEEE Access* 8 (2020), 52100–52112. doi:10.1109/ACCESS.2020.2979894

[15] Chrysoula Kroustalli and Stelios Xinogalos. 2021. Studying the effects of teaching programming to lower secondary school students with a serious game: a case study with Python and CodeCombat. *Education and Information Technologies* 26, 5 (2021), 6069–6095.

[16] Leslie Lamport. 1994. Introduction to TLA. *SRC Technical Note* 1994, 001 (1994), 1–7.

[17] Leslie Lamport. 2021. A high-level view of TLA+. https://lamport.azurewebsites.net/tla/high-level-view.html

[18] Leslie Lamport. 2022. TLA+ Tools. https://lamport.azurewebsites.net/tla/tools.html

[19] Leslie Lamport. 2024. PlusCal Tutorial. https://lamport.azurewebsites.net/tla/tutorial/intro.html

[20] James R Lewis. 1995. IBM computer usability satisfaction questionnaires: psychometric evaluation and instructions for use. *International Journal of Human-Computer Interaction* 7, 1 (1995), 57–78.

[21] Ju Long. 2007. Just For Fun: using programming games in software programming training and education. *Journal of Information Technology Education: Research* 6, 1 (2007), 279–290.

[22] Jan Lönnberg, Lauri Malmi, and Mordechai Ben-Ari. 2011. Evaluating a visualisation of the execution of a concurrent program. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '11)*. Association for Computing Machinery, New York, NY, USA, 39–48. doi:10.1145/2094131.2094139

[23] Sai Siddartha Maram, Johannes Pfau, Jennifer Villareale, Zhaoqing Teng, Jichen Zhu, and Magy Seif El-Nasr. 2023. Mining Player Behavior Patterns from Domain-Based Spatial Abstraction in Games. In *2023 IEEE Conference on Games (CoG)*. 1–8. doi:10.1109/CoG57401.2023.10333224

[24] Sai Siddartha Maram, Jennifer Villareale, Thomas Boyd Fox, Jichen Zhu, and Magy Seif El-Nasr. 2023. Parallel OPM: A Visualization System for Analyzing Peers Board States for Gameplay Reflection. In *2023 IEEE Conference on Games (CoG)*. 1–2. doi:10.1109/CoG57401.2023.10333145

[25] Tobias Mettler and Roberto Pinto. 2015. Serious Games as a Means for Scientific Knowledge Transfer—A Case From Engineering Management Education. *IEEE Transactions on Engineering Management* 62, 2 (2015), 256–265. doi:10.1109/TEM.2015.2413494

[26] Microsoft. 2025. Overview of the Visual Studio debugger. https://learn.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour?view=vs-2022

[27] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and reproducing Heisenbugs in concurrent programs *(OSDI'08)*.

[28] Chris Newcombe. 2014. Why Amazon Chose TLA + . In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Yamine Ait Ameur and Klaus-Dieter Schewe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 25–39.

[29] Jakob Nielsen. 1994. *Usability engineering*. Morgan Kaufmann.

[30] Santiago Ontañón, Jichen Zhu, Brian K. Smith, Bruce Char, Evan Freed, Anushay Furqan, Michael Howard, Anna Nguyen, Justin Patterson, and Josep Valls-Vargas. 2017. Designing Visual Metaphors for an Educational Game for Parallel Programming. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI EA '17)*. Association for Computing Machinery, New York, NY, USA, 2818–2824. doi:10.1145/3027063.3053253

[31] Stefan Resch and Michael Paulitsch. 2017. Using TLA+ in the Development of a Safety-Critical Fault-Tolerant Middleware. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 146–152. doi:10.1109/ISSREW.2017.43

[32] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[33] ruby concurrency. 2025. Class: Concurrent::Semaphore. https://ruby-concurrency.github.io/concurrent-ruby/1.1.5/Concurrent/Semaphore.html

[34] Bernd-Holger Schlingloff. 2021. Teaching Model Checking via Games and Puzzles. In *Formal Methods – Fun for Everybody*. Springer.

[35] Magy Seif El-Nasr, Elin Carstensdottir, and Michael John. 2026. *Designing Serious Games.* MIT Press.

[36] N.V. Shilov and K. Yi. 2001. Puzzles for Learning Model Checking, Model Checking for Programming Puzzles, Puzzles for Testing Model Checkers. *Electronic Notes in Theoretical Computer Science* (2001). doi:10.1016/S1571-0661(04)80893-4

[37] Ana Luisa Veroneze Solórzano and Andrea Schwertner Charão. 2021. BlocklyPar: from sequential to parallel with block-based visual programming. In *2021 IEEE Frontiers in Education Conference (FIE)* (Lincoln, NE, USA). IEEE Press, 1–8. doi:10.1109/FIE49875.2021.9637261

[38] Richard Stallman, Roland Pesch, Stan Shebs, et al. 1988. Debugging with GDB. *Free Software Foundation* 675 (1988).

[39] Filip Strömbäck, Linda Mannila, and Mariam Kamkar. 2022. A Weak Memory Model in Progvis: Verification and Improved Accuracy of Visualizations of Concurrent Programs to Aid Student Learning. In *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '22)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. doi:10.1145/3564721.3565947

[40] Tokio. 2025. Semaphore in tokio::sync. https://docs.rs/tokio/latest/tokio/sync/struct.Semaphore.html

[41] Stephanie Yang, Miles Baird, Eleanor O'Rourke, Karen Brennan, and Bertrand Schneider. 2024. Decoding Debugging Instruction: A Systematic Literature Review of Debugging Interventions. *ACM Trans. Comput. Educ.* 24, 4, Article 45 (Nov. 2024), 44 pages. doi:10.1145/3690652

[42] Zehui Zhan, Luyao He, Yao Tong, Xinya Liang, Shihao Guo, and Xixin Lan. 2022. The effectiveness of gamification in programming education: Evidence from a meta-analysis. *Computers and Education: Artificial Intelligence* 3 (2022), 100096.

[43] Jichen Zhu, Katelyn Alderfer, Anushay Furqan, Jessica Nebolsky, Bruce Char, Brian Smith, Jennifer Villareale, and Santiago Ontañón. 2019. Programming in game space: how to represent parallel programming concepts in an educational game. In *Proceedings of the 14th International Conference on the Foundations of Digital Games* (San Luis Obispo, California, USA) *(FDG '19)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. doi:10.1145/3337722.3337749

[44] Jichen Zhu, Katelyn Alderfer, Brian Smith, Bruce Char, and Santiago Ontañón. 2020. Understanding Learners' Problem-Solving Strategies in Concurrent and Parallel Programming: A Game-Based Approach. arXiv:2005.04789 [cs.HC] https://arxiv.org/abs/2005.04789

[45] Jichen Zhu and Magy Seif El-Nasr. 2021. Open Player Modeling: Empowering Players through Data Transparency. arXiv:2110.05810 [cs.HC] https://arxiv.org/abs/2110.05810

[46] Ilya Zverev. 2020. How to Catch a Cat with TLA+. https://medium.com/wavesprotocol/how-to-catch-a-cat-with-tla-b56fc2c75f59 Accessed: 2025-07-02.