An Entertaining Approach to Parallel Programming Education

Emanuel Buzek, Martin Kruliš
Parallel Architectures/Applications/Algorithms Research Group
Faculty of Mathematics and Physics, Charles University
Prague, Czech Republic
emanuel.buzek@gmail.com, krulis@ksi.mff.cuni.cz

***Abstract*—Despite the facts that multicore CPUs are present in virtually every personal computer or cell phone and distributed systems in the form of cloud services are steadily penetrating various domains of our lives, only a minority of programmers and computer science graduates are able to effectively design and develop parallel and distributed applications. Serial thinking is natural to all humans and it is also encouraged by many computer science curricula. Even though that leading educational institutions are attempting to rectify this trend by introducing parallel programming courses into their study programs, these courses are often dedicated for more experienced students in their fourth of fifth year since mastering modern parallel technologies like OpenMP or CUDA requires certain level of programming skills. It can be argued, that the parallel thinking should be taught much sooner, perhaps even before tertiary education. To this end, we have created an educational platform Parapple that aims to introduce parallelism and related problems like load balancing or synchronization to inexperienced programmers in an entertaining form. Our platform is web-based, so it can run in any modern browser on all operating systems without installation and the users are required to have only a very basic understanding of structural imperative programming.**

***Keywords*-parallel, programming, education, simple, visual, web**

I. Introduction

Parallelization has been one of the methods for achieving higher computational throughputs since the beginning of modern computer science. At the dawn of 21^{st} century, parallelism was widely employed in the domain of mainstream processors and commodity hardware components. Contemporary devices like PCs, laptops, or cell phones utilize multicore processors on regular basis. Furthermore, loosely coupled systems have grown into a new era of cloud computing and cloud become an integral part of various aspects of our lives.

However, parallel programming in the sense of designing applications for multicore, manycore, or distributed platforms is still a difficult discipline, perhaps even the most difficult domain of computer science. The problem is twofold: First of all, people are inherently sequential beings – i.e., we tend to think in sequential concepts and can fully focus only on a single task at a time. The second aspect of the problem lies in the hardware development itself as it proceeds really fast and neither the programmers nor programming course curricula are able to fully keep up with its pace.

Many contemporary educational institutions which provide computer science study programs place serious emphasis on parallel programming. On the other hand, they also consider parallel programming to be an advanced skill and relevant courses that cover multicore programming or middleware for distributed systems are situated later in the master courses. It may be argued, that the students need to have a good grasp of coding, algorithmization, and hardware fundamentals before they can be educated in parallel programming. On the other hand, postponing the introduction of parallel concepts emphasizes the importance of sequential concepts and the students often perceive parallelism as something unnatural and dangerous.

We have been working on a platform which would introduce the parallelism to coding beginners in a simple and entertaining way. It will offer short coding exercises which focus on various parallelization issues such as load balancing or synchronization. Even though the platform is not directly related to any existing parallel framework or technology, it tries to reflect the most important aspects and limitations of contemporary parallel hardware. Our main objectives are:

- *Simplicity.* The students should be able to start solving our educational exercises quickly and easily. The users are expected to have only basic coding skills, preferably in a C-like language or similar procedural or object-oriented language.
- *Attractiveness.* The exercises and their solutions should be presented in visually appealing and entertaining manner.
- *Versatility.* Parallel programming covers many software paradigms (data parallelism, task parallelism, fork/join model, etc.) and targets many hardware platforms (multicore CPUs with SIMD, manycore GPUs with lockstep execution, distributed systems with message passing, etc.). Exercises that simulate (possibly) all these paradigms and platforms should be supported.
- *Extensibility.* Adding new exercises and also new types of exercises should be supported in a reasonably simple way.
- *Accessibility.* The platform has to be available to broad spectrum of users on all mainstream platforms, prefer-

ably without installation (i.e., without elevated system privileges).

With respect to these objectives, we have designed and implemented a proof-of-concept web application *Parapple* [1] as a part of the master thesis [2] of the first author. The application introduces virtual environment for avatars – visualized automatons controlled by a script written by the student. The avatars may interact with each other and with the environment in order to solve objectives prescribed by the exercise scenario.

In this paper, we will present application Parapple and its concepts in more detail. Additionally, we provide several examples of exercises which correspond to real problems in parallel programming. The paper is organized as follows: Section II revises related work. Section III presents the architecture and principles of the Parapple application. Scripting language (ParaScript) used in Parapple is presented in Section IV and Section V covers the most important implementation details. Examples of parallel problems embodied in Parapple exercises are presented in Section VI. Section VII concludes our paper.

II. Related Work

The idea of converting learning concepts into games is not new. In fact, gamification is becoming increasingly popular especially for programming tutorials [3]. Some research has been done also in the domain of interactive in-class games that should introduce concepts from distributed and parallel programming [4]. In this section, we present some related work in both domains, with particular emphasis on the web platform.

A. Web Platforms for Coding Tutorials

Several wellknown web sites offer tutorials covering various topics of computer programming; Khan Academy (www.khanacademy.org) has a rich curriculum of computer science courses and their coding tutorials contain a very interesting feature – a video merged with an interactive coding editor. The student can change and run the code directly in the narrated video tutorial.

Another web site featuring broad range of coding tutorials is Code School (www.codeschool.com). It collects multiple courses covering several languages, programming tools, and databases.

Some web platforms focus on the gamification of coding. Notable examples are CodeCombat (codecombat.com) and CodinGame (www.codingame.com). Both platforms feature graphically appealing game environment and they offer several programming languages the students can use. In CodeCombat, the students learn basic programming topics by writing programs (in Python and JavaScript) for characters in an RPG game and the code changes are reflected in real-time. Similarly, CodinGame users solve short programming puzzles by writing a program in any of 20 supported programming languages.

Another elaborated puzzle-solving platform for programming exercises worth mentioning is CodeWars (www.codewars.com). It focuses on peer competitions and allows users to create new challenges. Popular platform for learning Python and JavaScript through exercises in browser is also CheckiO (checkio.org) and HackerRank (www.hackerrank.com).

There are several online projects that focus on so called code battles. The main principle is that the users write a program that competes against other programs of other users. For example, on Robocode5 (robocode.sourceforge.net) users program robot tanks in Java or .NET, and these tanks then engage in battles. Similar principle uses Fight Code (fightcodegame.com), where users create fighting robots by writing a JavaScript program. These battling platforms are especially interesting to look at in the context of parallel programming, as they introduce multiple competing agents.

However, neither of these platforms features tutorials truly cover parallelism. More importantly, although some of them may seem very extensible in terms of graphical representations of the game, they are not suited for processing of multiple programs in parallel.

B. Platforms for Parallel Programming Tutorials

Deadlock Empire (deadlockempire.github.io) is an online project that teaches concurrency topics by giving the user the role of a scheduler. In each tutorial, the user is presented with two or more short programs in C#. The user then steps through them concurrently by choosing which program executes next in every step. The objective is to make the programs crash – for example, cause a deadlock or enter a critical section in more threads simultaneously. While such platform is interesting for demonstrating some topics in concurrency, it does not really teach solving programming tasks with parallel approach and its educational scope is somewhat limited.

A more specific tutorial was presented by nVidia. It is intended to tach GPU programming without actually having a GPU. This online CUDA tutorial platform is fully available in browser and allows users to explore several topics including OpenACC, Multi-GPU programming, and GPU Memory Optimizations [5].

Despite the fact that first parallel programming tutorials have emerged, we still believe that this domain is not sufficiently represented. Especially when focusing on coding beginners.

III. Parapple Application

Our Parapple application was implemented as a proof of concept to demonstrate our proposed approach to education

of parallel programing principles and fundamentals. The application is composed of two main components – virtual environment for avatars and simple integrated IDE which allows writing the controlling scripts and debugging them. Once an exercise is opened, the virtual environment visualizes the initial setup which is typically a 2D grid with avatars and other additional elements (tokens, walls, checkpoints, etc.). The user can use the IDE to write a script for each group of avatars (avatars in a group share the same program) and then let the environment simulate the avatars using these scripts.

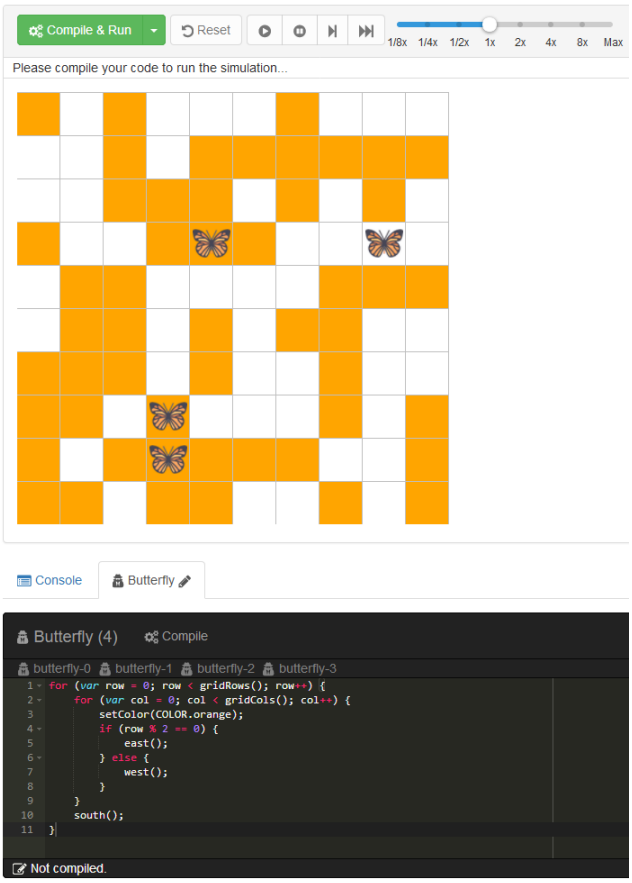

Figure 1. Parapple workbench (virtual environment and code IDE)

Each exercises has a predefined objective which the avatars need to achieve. Once this objective is reached, the simulation terminates and the user has completed the exercise successfully. The user may also pause the simulation and debug it step by step whilst watching the state of all avatars (including their variables). The user may also reset the simulation when it is obvious the avatars are not progressing towards objective completion.

In order to make the whole platform as versatile as possible, the virtual environment provides only a minimal set of functions. In fact, it only provides a single HMTL5 canvas elements with no API bindings for the avatars. All other functions are defined in extensions called *packages*.

A. Extension Packages

An extension package is a collection of JavaScript modules and other resources (e.g., images) which can be used in visualization. Each package export three collections of assets:

- Avatar type definitions, which are basically classes used for creating avatar objects. Avatar type is typically bound with a set of operations it can do.
- Functions and constants which are included to avatar API (i.e., which can be called and accessed by avatar scripts).
- Internal functions which can be used by other packages.

Furthermore, a package provides a list of dependencies – other packages that need to be loaded before the package can be used. These dependencies simplify code decomposition as one package may be built on top of another.

It would be tedious to create visually attractive exercises directly on the top of canvas element. Hence, two basic visualization packages were implemented – **pixi** and **pixi2d**. The former introduces a graphical JavaScript library of the same name[1], so the drawing on the canvas may be realized more effectively. The latter extends the functionality of the former by defining an environmental layout as 2D grid of square tiles into which object and avatars are placed. This grid is suitable for most exercises, but the modularity of this design allows creating different layouts (e.g., pixi-hex for hexagonal grid) if need be.

B. Exercises

Packages are used to construct *exercises*. More precisely, an exercise is provided with a list of packages it requires, a specification (text describing the exercise objective and other instructions for the students), initial setup function which sets the environment and create avatars, and an objective function which is tested after every step of the simulation to determine, whether the exercise should be successfully terminated.

En exercise is a single problem that is expected to be solved by the user. We have designed our initial set of exercises to demonstrate the functionality of the platform and we describe model exercises for various typical issues of concurrent processing in Section VI. However, since the packages can be shared among exercises and the most common avatars and APIs for their interaction were already implemented, creating new exercises should be very simple for anyone, who can code in JavaScript.

IV. ParaScript Language

Selecting appropriate scripting language for the avatars was one of the most difficult decisions of this project.

[1] http://www.pixijs.com/

Since the script has to be compiled and interpreted in the browser, we have decided to implement the compiler ourselves. Implementing a fully-working compiler for existing mainstream language is a very complex task. Furthermore, a scripting language for controling avatars does not require all features of modern programming language. Hence, we have decided to design our own language as a simplification of an existing language.

Considering the language properties:

- Structured imperative language is prefered over declarative and functional languages. The main reason is that declarative and functional languages typically hide many execution details including potential concurrent processing or synchronization. Furthermore, beginners often start coding courses in imperative languages.
- Procedural language is prefered over object-oriented language. Even though object-oriented languages are typical representants of mainstream imperative languages, our coding exercises are intended to be simple and short, so procedural decomposition is more than sufficient. We also need to consider the scope of the compiler as we have decided to implement it ourselves in the browser.
- Dynamic typing is prefered over static typing. In this case, there are strong arguments for both possibilities. The dynamic typing was selected since it is simpler for implementation and might be easier to use, especially if the intended scope is to write short scripts.

As a result, we have decided to base our new language ParaScript on JavaScript. JavaScript is a very popular language [6] with C-like syntax so it should be easy enough to learn. It will also simplify the script interpretation as some constructs of the language (especially data values) could be directly mapped into JavaScript constructs and processed in the browser natively.

A. Syntax and Language Properties

Detailed syntax overview is well beyond the scope of this paper. However, since ParaScript is based on JavaScript, we focus mainly on the differences of the two languages (more detailed reference is provided in related work [2]). There are three main differences:

- ParaScript is only procedural language whilst JavaScript is object-oriented language. This modification was done by simply omitting all object-related features of the language.
- JavaScript has first-class functions (i.e., functions are dynamically created values like objects). ParaScript has third-class functions and forbids nesting function declarations (i.e., functions are resolved at compile time and they cannot be created dynamically nor called indirectly).
- JavaScript has both function-scoped variables (declared with `var`) and block-scoped variables (declared with `let`). ParaScript has only block-scoped variables and uses only keyword `var` for the declaration.

The ParaScript has the same basic data types as JavaScript, except for `object` and `function`. Since objects were essential for creating structures and arrays, we have added `array` and `map` data types. Arrays are basically ordered lists and they inherit most behavior from JavaScript arrays – except for methods since arrays are not objects. Maps are key-value dictionaries which are intended as lightweight replacement for objects. They share basic syntax with JavaScript objects (construction, accessing properties), but they do not use prototyping and they cannot hold functions as properties (i.e., they do not have methods).

Finally, there are no predefined functions specified in the ParaScript language. However, Parapple implements core function package which comprises also some functions provided in the JavaScript fundamental objects (e.g., math functions, array and map functions, or sting functions).

B. Compilation and Intermediate Code

The ParaScript code si compiled into intermediate code which is then interpreted by avatar simulator. The simulator itself is a simple stack-machine with 1-operand instructions. This design may be suboptimal from the performance point of view, but stack-machines are well established models which are straightforward to implement. Therefore, we can focus on the parallel scheduling of concurrent stack-machines instead of the complexities of more advanced execution models.

The instruction set comprises standard operations for flow control, stack manipulation, and basic unary and binary operators. The instructions themselves are not represented in a binary form, but rather in form of JavaScript object, which simplifies processing and allows more versatile interpretation. For instance, we have only one instruction for all binary operations and its operand determines the exact operation (whilst the data operands are taken from the stack).

The compiler design follows widely established technologies. It is based on GNU bison[2] LALR parser generator for context-free languages and Flex[3] lexical analyzer. However, since our implementation was designed to operate solely in the web browser we have decided to use a JavaScript implementation of these two tools called Jison [7].

V. Implementation Details

In this section, we will describe the most important implementation details. We will focus mainly on the virtual machine and the instruction scheduler, which are essential for the simulation of avatars.

[2]https://www.gnu.org/software/bison/

[3]https://github.com/westes/flex

A. Instructions

As mentioned in the previous section, the instructions are not represented by a binary code as it would require unnecessary encoding and decoding. We use regular JavaScript objects instead, since both the compiler and the interpret (virtual machine) are implemented in JavaScript. This means that the code itself will take much more memory; however, the whole solution is intended for short scripts so the real impact on memory consumption is negligible.

Each instruction object comprises:

- *Instruction code* which uniquely identifies instruction type.
- *Operand* represented as JavaScript value of any type (actual type depends on the instruction).
- *Execution function* which is responsible for performing the instruction.
- *Cost function* used for the calculation of instruction processing time.
- *Source location* containing the line and column of the source code (for debugging purposes).
- *Breakpoint* – a flag which determines whether a breakpoint was set before this instruction.

The execution function is a regular JavaScript function which implements the instruction execution. All instructions of a given type refer to the same function.

The cost function determines the cost of the instruction, which is an integer. It is passed along with the instruction so it can be overridden by exercises to approximate different hardware platforms. The cost value is not a constant in order to accomodate more complex cost models and to allow random values which could simulate nondeterministic nature of many parallel systems. The cost returned from the function is reflected by the scheduler which delays the execution of the following instruction accordingly.

B. Virtual Machine and Instruction Schedulers

The virtual machine is a regular stack machine with 1-operand instructions. In fact, we are using three separate stacks:

- *Main stack* is used for arithmetic operations and argument passing.
- *Address stack* is used for return addresses when user functions are called.
- *Memory stack* holds local variables (a JavaScript object is pushed to this stack with every function call and it acts as a namespace for the variables[4]).

Each avatar has its own instance of the stacks and each avatar group share the same program. This way each exercise may choose whether each avatar has its own program (which tends to simulate task parallelism) or whether some (possibly all) avatars share the program (which tends to simulate data parallelism).

The virtual machine is wrapped with *executor*. It holds the complete state of the avatar which mainly comprises the three stacks, the instruction pointer, and a reference to the code the avatar executes. In one *step*, the executor takes the following instruction of its code and executes it. After that, it invokes the cost function of that instruction to determine execution *penalty*. If the penalty equals zero, it immediately executes the following instruction. Otherwise, the value of penalty indicates, how many following steps should the executor skip.

Executors are being invoked by a *scheduler*. In each *tick*, the scheduler invokes all active executors, so they can perform one step. We have implemented three types of schedulers:

- *Deterministic scheduler* invokes the executors in a round robin fashion in fixed order.
- *Nondeterministic scheduler* invokes the executors in random order.
- *Lock-step scheduler* enforces the executors to execute the same instruction at a time.

Standard schedulers (deterministic and nondeterministic) tend to simulate the environment of regular multicore systems and distributed systems. The lock-step scheduler is designed to simulate SIMD operations or SIMT lock-step execution which is typical for GPUs. The lock-step scheduler is much more complex than the standard schedulers, so we describe its principles in more detail in the following section.

C. Lock-step Scheduler

In the lock-step execution model, all execution units perform the same instruction at a time. The greatest problem of this model is code divergence – i.e., divergence in a code flow caused by a conditional branches such as `if` or `while` statements. This problem is typically solved by instruction masking. All execution units (in our case executors) follow all code branches, but they mask instruction execution based on their local condition evaluations.

In our case, the branching problem is caused by conditional jump instruction. The solution of this problem takes an advantage from a simple fact that the compiler generates conditional jumps so that the target address is always greater than the position of the instruction (i.e., it always jumps ahead). The main idea is that the executors that perform the jump are removed from the active set and wait until the collective instruction pointer reaches the target address of the jump.

The situation is slightly more complicated since the control structures that generate conditional jumps may be nested. Therefore, we have a global stack of sets of suspended executors and when a conditional jump occurs, the jumping set of executors is pushed to the stack. Furthermore, each suspended set is associated with the target address of

[4]Please note that parascript has block-scoped variables; however, the block-scope visibility is resolved at compile time.

the jump so we can easily determine when is the right time to reactivate the set.

There are two special cases, which need some more attention – the `else` statement and `break` and `continue` statements. Both these cases are translated into unconditional jumps, but they need to be resolved differently. To handle these statements correctly, we have modified the compiler to mark each jump instruction by the original statement for which it was generated.

The `else`-statement jump basically requires that the current active set of executors is swapped with the waiting set on the top of the stack. In other words, currently active executors are suspended and the executors suspended at the last conditional jump (which must have been `if`-statement) are reactivated.

The `break` and `continue` statements are even more complicated. Even though they are related to the nearest loop construct, they may be nested in (possibly many) `if` statements. The executors that encounter these statements has to be suspended and added to the waiting sets at the end or at the beginning of the nearest loop respectively.

Finally, we need to address the issue of interoperability. The lock-step model simulation attempts to create the appearance that the executors perform the issued instruction simultaneously. However, since the scheduler is sequential, it invokes one executor after another. In order to allow implementation of cooperating functions (like GPU warp-shuffle operations), the scheduler provides a hook for callbacks. The instruction may provide a callback function which is invoked at the end of the tick and it can be used to alter the internal result of the instruction.

D. Interoperability

The executors are completely isolated and may not interact with each other nor with the outside environment. Such interaction must be provided explicitly by the functions in the packages. Package functions can be called by special instruction. Calling arguments are passed via stack using the same convention as for user function calls.

The package functions are regular JavaScript functions and they can perform any task. The two typical categories are avatar interactions with the environment (movement, interaction with the grid, interaction with tokens, etc.) and avatar interaction with other avatars (detection, communication). This way, each exercise may provide API tailored to its specific needs.

E. Other Technical Details

The Parapple is implemented as single-page application (SPA), since traditional cgi-like application would have negative impact on the user experience. It means, that the whole solution runs in a browser without additional support from the server. On the other hand, it would be possibile to add additional features like user-history which can be persisted on the server if need be.

There are many frameworks for designing single-page web applications. We have selected Vue [8], which is modern lightweight framework. However, other mainstream frameworks like Angular[5], React[6], or Ember[7] would be also a good choice.

VI. Problem Examples

In this section, we present problem examples which can be implemented as exercises in Parapple. The problems are explained in the terms of exercise specification and we point out the lesson the student should learn by solving it. All the following exercises operate in a virtual environment which is represented by a 2D grid. Avatars can move on the grid and there may be at most one avatar on a tile. Furthermore, each tile of the grid may have additional properties (e.g., color) and appropriate API is provided so that avatars may move on the grid and interact with these properties.

A. Counting Tiles (Reduction)

In this scenario, the tiles of the grid are colored randomly (white and black). The avatars may determine the color of the tile they are standing on and the objective is to cooperatively determine the number of black tiles. The data exchange is performed also using the tiles as each tile have an associated integer and avatars may read and write the number associated with the tile they are standing on. At the end, each avatar must have the final result.

This is a typical exercise for parallel reduction. The data exchange is slightly more complicated; however, it follows the shared memory principle with the exception that the synchronization is enforced explicitly by the position. The scenario should also reflect some properties of cache coherency protocols (e.g., MESI) which are often employed in shared memory architectures.

B. Tile Coloring (Load Balancing)

The grid tiles are initially colored all black and the objective is to change them all to white. The avatars may determine and change the color of the tile they are standing on, but they have no additional means of communication. The initial positions of the avatars are set randomly. Finally, the instruction costs (especially the cost of changing the tile color) are different for each avatar; therefore, some avatars work significantly faster than others.

This exercises focus on work load balancing. As the positions are random and each avatar has its own speed (which could simulate heterogeneous systems or process interference on a saturated system); thus, a dynamic load balancing strategy has to be employed (e.g., task stealing).

[5]https://angular.io/
[6]https://reactjs.org/
[7]https://www.emberjs.com/

Furthermore, some simple synchronization using only the tile colors has to be implemented to avoid redundant coloring.

C. Row Coloring (Lock-step Demonstration)

Similarly to the previous exercise, our objective is to adjust colors of selected tiles. However, in this scenario, each row is occupied by exactly one avatar, which is initially situated on its leftmost field. The avatars traverse their rows from left to right in a coordinated manner. On each column, they cooperatively determine the dominant color of the column and make it the only color. There are several variations of this exercise, for instance:

- *Communication API* – The avatars may exchange messages with their neighbours only, or they may have more sophisticated means like broadcast function.
- *Scheduler* – The execution can be performed in lock-step (which actually simplifies this task) or using one of the standard schedulers. In case of standard scheduler, the exercise become most interesting if each avatar has a different instruction cost (especially the cost of moving).

This exercise is primarily ment for introducing the lock-step execution model and its nuances. Based on the communication API restrictions, it may be also used to demonstrate the workgroup-wide functions or as a training in complex communication patterns.

D. Tile Coloring with Masters and Workers

We may explore the topic of tile coloring from yet another angle. In this last variation of coloring exercise, two types of agents are introduced – *masters* and *workers*. Masters are fast and they can read the color of any tile on the grid. However, the cannot change the colors. Workers can change the color of the field they are standing on, but they move very slowly and they can read only the color of their field. Masters and workers can exchange messages if they stand on adjacent tiles. The objective is to convert the entire grid to white.

This exercise aims to simulate some issues of a distributed system with complex communication middleware. The masters can be roughly related to communication logic. The workers are individual hosts of the system that do the actual work.

E. Mining (Sharing a Resource)

This final example is inspired by real-time strategy games, where the players collect some form of resources. In this scenario, we have a gold mine that is represented by a narrow passage which has gold nuggets on its last tile. Only one avatar can reach the gold at a time as two avatars cannot avoid each other in the passage. The objective is to transport all nuggets to specific location on the grid. The avatars cannot communicate, but they can place marks on tiles.

The exercise demonstrates operations with a shared resource. A means of synchronization using tile marks has to be devised so that only one avatar enters the mine at a time. Furthermore, the avatars must form a queue at the entrance whilst not blocking the path of the exiting avatar.

VII. Conclusions

In this paper, we have presented a new educational platform Parapple. Parapple is a single-page web application that attempts to present simple coding exercises which aim explicitly to teach the concepts and perils of parallel and distributed programming. The user is expected to write short scripts (in our own language ParaScript) that control avatars, which can perform tasks that are conceptually related to work performed by individual CPU or GPU cores, nodes in distributed system, or even tasks hidden in the hardware or communication middleware when running a parallel application.

Even though we have tested the application on limited number of users which can be hardly statistically significant, the concepts introduced by Parapple triggered positive responses and constructive feedback. In our future work, we will focus on exposing this platform to a wide variety of users from coding novices to graduate students in order to collect more relevant evaluation data. Furthermore, we are working on a wider pallette of exercises to cover more aspects of parallel and distributed computing.

Acknowledgment

This work was supported by project PROGRES Q48.

References

[1] E. Buzek. (2017) Parapple. a platform for parallel education. [Online]. Available: http://www.ksi.mff.cuni.cz/en/sw/parapple.html

[2] E. Buzek, "Web platform for parallel programming tutorials." Univerzita Karlova, Matematicko-fyzikální fakulta, 2017.

[3] D. Pranantha, C. Luo, F. Bellotti, and A. de Gloria, "Designing contents for a serious game for learning computer programming with different target users," 2011.

[4] R. F. Maia and F. R. Graeml, "Playing and learning with gamification: An in-class concurrent and distributed programming activity," in *Frontiers in Education Conference (FIE), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[5] NVIDIA. (2014) Learn GPU programming in your browser with NVIDIA hands-on labs. [Online]. Available: https://devblogs.nvidia.com/learn-gpu-programming-free-on-demand-gpu-training/

[6] (2018) GitHut 2.0: GitHub language statistics. [Online]. Available: https://madnight.github.io/githut/

[7] Z. Carter. (2012) Jison. [Online]. Available: https://zaa.ch/jison/about/

[8] Vue.js. comparison with other frameworks. [Online]. Available: https://vuejs.org/v2/guide/comparison.html