Parallel Programming with Pictures in a Snap!

Annette Feng* and Wu-chun Feng*†
*Department of Computer Science
†Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, U.S.A.
{afeng, wfeng}@vt.edu

***Abstract*—For decades, computing speeds seemingly doubled every 24 months by increasing the clock speed and giving software a "free ride" to better performance. This free ride, however, effectively ended by the mid-to-late 2000s. With clock speeds having plateaued and computational horsepower instead increasing due to increasing the number of cores per processor, the vision for parallel computing, which started more than 40 years ago, is revolution that has now ubiquitously arrived. In addition to traditional supercomputing clusters, parallel computing with multiple cores can be found in desktops, laptops, and even mobile phones. This ubiquitous parallelism in hardware presents at least two major challenges: (1) difficulty in easily extracting parallel performance via current software abstractions and (2) difficulty in delivering correctness — as even without parallelism, software defects already account for up to 40 percent of system failures. Consequently, this paper presents preliminary research that reduces the learning curve to parallel programming by introducing such concepts into a visual (but serial) programming language called Snap!. Furthermore, the proposed visual abstractions automatically generate parallel code for the end user so as to better ensure that the resulting (text-based) code is correct.**

***Keywords*—explicit parallel computing; computer science education; block-based programming; visual programming; parallel computational patterns**

***EduPar Topical Area*—pedagogical tools, programming environments, and languages for PDC and HPC**

I. Introduction

As complex (or higher-order) reasoning skills are now driving advanced economies, as shown in Figure 1), manual tasks and routine cognitive tasks are being increasingly automated. As a result, higher-order skills requiring complex reasoning and communication must become a major focus of educational strategies. Indeed, the College Board, in partnership with NSF, recently announced the fall 2016 launch of their new Advanced Placement Computer Science Principles course. In development since 2009 with funding from NSF, the AP Computer Science Principles course "is designed to broaden the number and diversity of students who participate in computing" and to empower them to "develop skills that will be critical to the jobs of today and tomorrow" [1], [2].

Many of these higher-order reasoning skills can be acquired in the context of computing. Because computing has emerged as a third pillar of science, complementing the traditional pillars of theory and experimentation, it can accelerate discovery and innovation and create a fundamental change in how research, development, and technology transfer in the sciences, engineering, business, humanities, and arts will be conducted in the 21st century. For example, in a study conducted by the U.S. Council of Competitiveness in 2004, 97% of surveyed U.S. businesses noted that they could *not* exist or compete without the innovative use of high-performance parallel computing (HPC) [3]. Unfortunately, those same companies lament the dearth of a trained workforce that is familiar with parallel computing concepts.

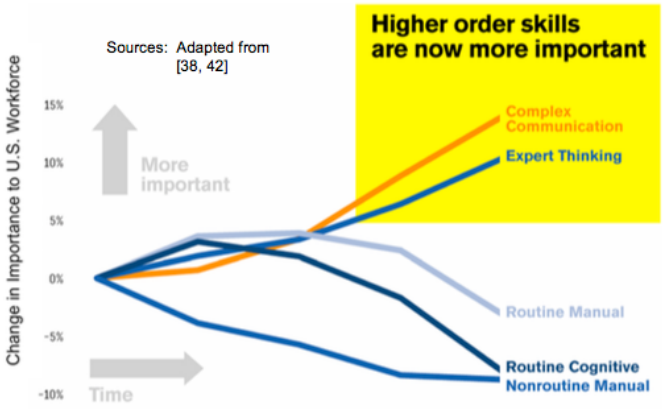

Fig. 1. Technological Changes Affecting U.S. Workforce Skills.

More recently, we have seen parallel computing become ubiquitous. For decades and until the mid-to-late 2000s, computing speeds seemingly doubled every 24 months by increasing the clock speed and giving software a "free ride" to better performance. With clock speeds having plateaued and computational horsepower instead increasing due to increasing the number of cores per processor, parallel computing is now the norm. In addition to traditional supercomputing clusters, parallel computing with multiple cores can be found in desktops, laptops, and even mobile phones. This ubiquitous parallelism in hardware presents at least two major challenges: (1) difficulty in easily extracting parallel performance via current software abstractions and (2) difficulty in delivering correctness — as even without parallelism, software defects already account for up to 40 percent of system failures. To address these challenges, we turn to block-based programming environments.

Block-based programming environments, such as Scratch [4] and Snap! [5], have been used effectively as powerful educational aids to introduce beginners to computing. We see the broad appeal of these environments due to the following two features. First, block-based languages have a very low barrier to entry. That is, students with no prior programming experience can quickly grasp the skills required to build programs that capture their interest, thereby motivating them to keep learning how to program. Second, block-based programming scales well with respect to students' ages and their level of programming experience. Block-based languages are expressive enough to support the ingenuity of quite advanced students of computing, while still providing enough basic blocks to provide a rewarding programming experience to novices. It is because of these properties of block-based languages that we see them as fertile ground for introducing parallel computing concepts to a wide range of computing students.

To address the need of improving the teaching of parallel comput-

978-1-4799-5944-0/14/$31.00 ©2014 IEEE

ing concepts, we seek to add *explicit* parallel abstractions to block-based programming languages. The thesis of this research is that the teaching of parallel computing does not need to be postponed until students have mastered the fundamentals of sequential programming. In fact, at this point, it may be too late to groom students to think truly in parallel. Instead, we posit that explicit parallel abstractions, such as producer-consumer, should be viewed as fundamental to programming as the `for` loop. By exposing explicit parallel programming via key language abstractions, we aim to harmoniously introduce students to parallel computing from the very start.

The rest of the paper is organized as follows. Section II covers background including discussion of the Snap! programming environment and concurrency paradigms. Section III presents work we've done to demonstrate the viability of this approach. Finally, in Section V we present our conclusions and future work.

II. Background

Snap!, based on the Scratch programming language developed at MIT, is a "drag-and-drop" visual programming environment in which end users build programs that control the behavior of actors called *Sprites*. Snap!'s look and feel is very similar to that of Scratch. However, whereas Scratch was originally written in Smalltalk as a standalone application, Snap! is written in JavaScript to run in a web browser. Snap! also includes several key features and capabilities not found in Scratch that we introduce later in this section.

Figure 2 shows what the Snap! interface looks like. The white *stage* area in the upper right of the interface is where the sprites appear and display their output. A typical Snap! project consists of multiple sprites along with their scripts which specify their actions. Each sprite has its own collection of scripts that appears in the *scripts* area in the center of the interface. The interface always points to a current sprite whose scripts are the ones displayed for editing. The user switches between sprites to edit each one in turn. In the example shown in Figure 2, the project consists of two sprites, a cat and a bat, with the bat being the current sprite with a program consisting of two separate scripts.

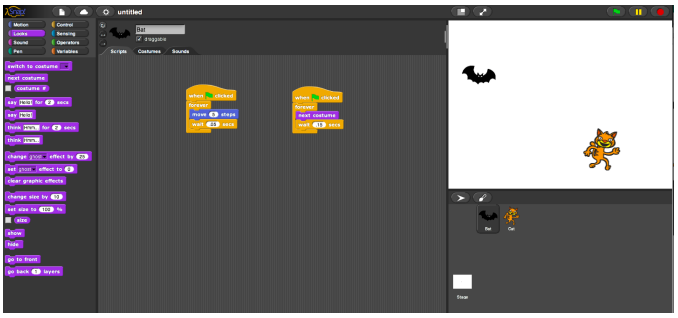

Fig. 2. The Snap! Graphical User Interface.

Users create programs by dragging blocks from the *palette* area on the left side of the interface and assembling them in the scripts area. The blocks in the palette are grouped by category, with one category of blocks being displayed at a time according to the current selection. The color, style, and shape of the blocks determine how they may be assembled to create valid programs. Blocks "snap" together and nest inside each other to form scripts, and each sprite can contain any number of scripts that collectively determine its behavior when the project is run. The bat sprite has two scripts that are shown in Figures 3 and 4. One script is for translating the sprite across the stage and the other is for continually toggling the appearance of the bat between wings up and wings down. When the user runs the project

Fig. 3. Move process.

Fig. 4. Flap wings process.

by clicking on the green start button, both scripts run simultaneously and the bat appears to "fly" across the stage flapping its wings. This is depicted in Figure 5 as a series of visual time steps.

Fig. 5. Bat sprite appearing to fly across the stage.

This form of parallelism is referred to as *concurrency*. Because JavaScript is single-threaded, the illusion of parallelism in Snap! is achieved through *multi-tasking*. Multi-tasking is a technique for executing all active processes one at a time in an interleaved fashion with only a single thread of control. When a Snap! project is run, all processes that match the criteria for starting, in this case, those scripts beginning with a "`when green flag clicked`" block, are added to the process queue by Snap!'s thread manager. Each process executes for a short amount of time called a *time slice* before yielding to the next process and waiting for its next allotted time slice. Because the computer can switch rapidly from one process to the next, this gives the illusion of parallel execution. In this manner, the interleaved execution of the bat scripts results in the visual effect of a bat flapping its wings as it flies across the stage.

This concurrent programming model exemplifies a form of *implicit* parallelism, as the creation and execution of Snap! processes is automatically managed by the underlying Snap! implementation, the details of which are all hidden, without the end user having to be aware of any kind of parallel constructs. Although Snap! is inherently concurrent, it does *not* teach or promote *explicit* parallel programming. In addition, because the entire execution of Snap! scripts occurs within its single browser thread, computationally expensive scripts can slow down the execution of Snap! programs, as well as render the browser unresponsive.

The specification of HTML5 Web Workers [6] is a way of addressing some of the problems arising from JavaScript's single-threaded nature. Web Workers provide a method for long-running JavaScript programs to spawn separate background threads that can utilize the underlying multi-core architecture of the host system, heretofore ignored by JavaScript programs. These background threads run independently from any user-interface threads and also independently from each other. One limitation of HTML5 Web Workers is that they cannot be used to perform work on user-interface elements. However, a situation in which they *can* prove useful is demonstrated in the following example.

Figure 6 shows a Snap! block called `map` that applies the function supplied in the first input slot to each element of the list supplied in the second input slot and returns a new list containing the results. The results of executing this code fragment are shown in Figure 7. Unlike

Fig. 6. Snap!'s map block.

Fig. 7. Results of `map(x 10)`.

in Scratch, lists and functions are both first-class data elements in Snap!, a key feature which, among other things, lets them be passed to and returned from procedures.

The `map` function executes serially by looping over a list, applying the supplied function to each list element, and ultimately returning a new list containing the results. Upon closer inspection of the `map` block of Figure 6, we see that it contains the binary multiplication operator with its first input being empty and its second input containing the number 10. The empty input signals where the list inputs are to be inserted into the function. Because the multiplication function is supplied as an argument to the map function, it would normally be evaluated first in order to obtain the input value to its enclosing block, in this case the `map` block. This is because Snap!'s default behavior is to evaluate all block arguments first, then evaluate the block. However, the function itself is the desired input value as it must be repeatedly evaluated with a different input element from the list each time. Therefore, the evaluation needs to be delayed until elements of the list are inserted, and the final results are stored in a dynamic list. The gray ring around the multiplication block signals for Snap! to delay the evaluation, hence causing the multiplication function itself to be treated as the input parameter to `map` instead of its value, which would simply evaluate to 0 given the empty input slot. Having functions as first-class elements allows them to be passed as arguments to other functions, to be assigned to variables, and to be returned as results.

This example shows the potential where HTML5 Web Workers can be utilized within the Snap! environment, as the computation involves mathematical operators and no user interface elements. For complex, user-defined computations, our Parallel Snap! can provide an ideal introduction to parallel programming for beginning programmers.

In the following section, we discuss the design and implementation of our parallelized Snap! environment which simultaneously leverages HTML5 Web Workers as well as Snap!'s first-class lists and procedures, and another key feature, *code mapping*, that programmatically translates Snap!'s visual block-based scripts to text-based code that can be exported and run externally.

III. Approach

Our approach to introducing true parallelism to Snap! draws inspiration from OpenMP [7], [8], an application programming interface (API) that supports multi-platform, shared-memory, multiprocessing programming across a multitude of programming languages (e.g., C, C++, and Fortran), operating systems, and processor architectures. The OpenMP API consists of a set of compiler directives, library routines, and environment variables that enable parallel execution at run time.

OpenMP is a relatively simple, *text-based* approach that introduces parallelism into a sequential program. Here is a simple sequential C program to print out "hello(0), world(0)":

```
void main()
{
  int ID = 0;

  printf(" hello(%d), ", ID);
  printf(" world(%d) \n", ID);
}
```

By adding a simple directive (or pragma) and a function call to obtain the thread ID, the following code readily compiles into a parallel program, where each thread prints out its own "hello(%d), world(%d)" message, where %d is the thread ID.

```
#include omp.h

void main()
{
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();

    printf(" hello(%d), ", ID);
    printf( world(%d) \n, ID);
  }
}
```

This is in stark contrast to the complexity of other text-based approaches, such as pthreads.

We seek to emulate the simplicity of text-based OpenMP parallelism using the `pragma` approach, but with a block-based approach instead. With our research extensions to Snap!, we introduce new built-in blocks to support parallelism on the front end, i.e., as a parallel programming API to the end user, and to generate any target text-based language on the back-end. We accomplish this by leveraging Snap!'s *built-in code-mapping features* to generate the desired text-based code, which can then be compiled and run on the target back-end system (including the Snap! environment itself).

Figure 8 introduces our new built-in `parallelMap` block, which serves as a visual equivalent to the text-based OpenMP `omp parallel` pragma block. This `parallelMap` block integrates the visual representation of Snap!'s `map` block with a back-end implementation that utilizes explicitly parallel HTML5 Web Workers, instead of Snap!'s serial execution model. The `parallelMap` block looks essentially the same as the original built-in function, but the underlying implementation makes use of a key feature of Snap!: *codification support*.

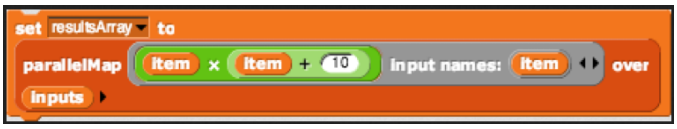

Fig. 8. `ParallelMap` function.

Codification support in Snap! is an experimental feature that is used to translate the visual, block-based programs of Snap! into any text-based programming language [9]. Figure 9 shows a small sampling of all the mapping constructs needed to translate Snap! blocks into JavaScript code.

Figure 10 shows a Snap! implementation of the `map` example from Figure 6. In this example, the map operation is written explicitly in long form so that the code translation is easier to follow.

The Snap! `code of` block, when executed, automatically translates the script it points to into the corresponding JavaScript code shown in Figure 11, according to an internal mapping that the

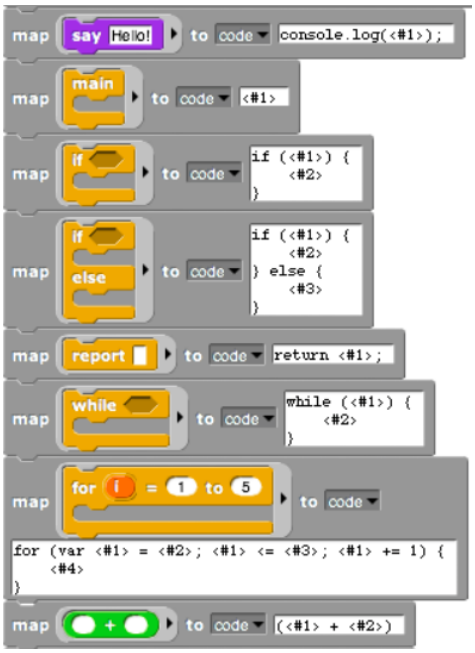

Fig. 9. Portion of Snap! to JavaScript code mapping.

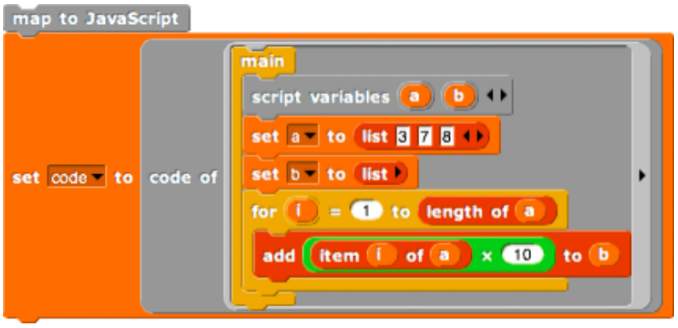

Fig. 10. Snap! script to map to JavaScript.

user specifies. The `map to JavaScript` block at the top of the script, must be executed first to set the internal code mapping to the JavaScript programming language so that the subsequent execution of the `code of` block does the code translation correctly.

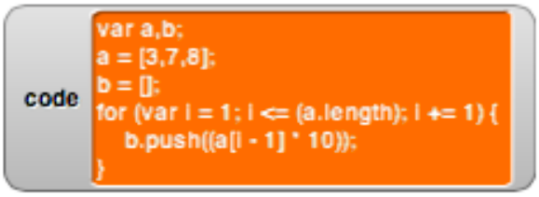

Fig. 11. Code mapping to JavaScript.

To change the back-end language to which the Snap! scripts are being mapped, i.e., if the user wishes to switch from JavaScript to C, (s)he simply changes the `map to JavaScript` block to a `map to C` block that contains the mappings appropriate for generating the C code shown in Figure 12.

It is the mapping to JavaScript capability that we utilize in order

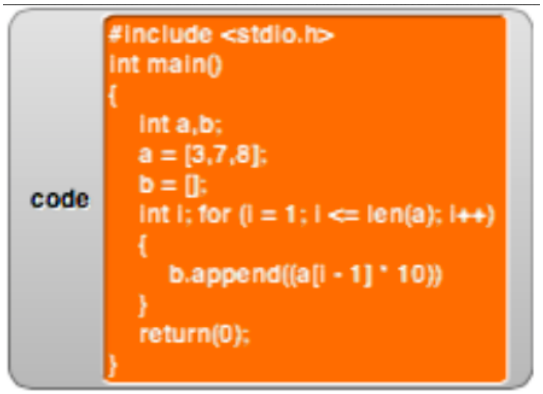

Fig. 12. Code mapping to C.

to generate the back-end code necessary to integrate our solution with HTML5 Web Workers, permitting true parallel execution using Snap! as a front end. Parallel.js is a small open-source JavaScript library that can be integrated into any JavaScript project simply by loading it in the project's .html file. Parallel.js provides a simple, straightforward API to HTML5 Web Workers [10]. Figure 13 shows how easily the library can be used to create Web Workers.[1]

```
function mydouble(n) {return n+n;};

var p = new Parallel([1, 2, 3, 4], {maxWorkers: 2});

p.map(mydouble);
console.log(p.data);
```

Fig. 13. Example code using Parallel.js.

In this example, we wish to take each element of the input list and return its double, the function for which is supplied as `mydouble`. The parallel job `p` is first created by calling `new Parallel` and supplying the list over which the workers are to operate. The optional argument to the `new` operator specifies the maximum number of Web Workers to use, which defaults to the number of cores or 4. In this example, two Web Workers are spawned upon job creation, with each worker receiving a copy of the `mydouble` function and the unique element of the list upon which it is to operate. When all the workers complete and the entire list has been processed, the result can be retrieved in the parallel object's data property. This example shows Web Workers operating on a hard-coded function, but with Snap!'s code mapping feature, we are not limited to hard-coded functions; Web Workers can be passed any function that the user creates on the fly in the Snap! interface as long as the translation for each Snap! block into JavaScript is defined. The underlying system takes care of the rest.

The new `parallelMap` block introduced in Figure 8 has an additional input slot that the user can reveal by clicking on the rightmost right-facing arrow next to the "inputs" array. This input allows the user to specify the number of Web Workers to spawn. Figure 14 shows two lists corresponding to this example of the `parallelMap` block. The one on the left shows the first ten inputs of the original input list to the `fig:parallelMap` block in the example, and the one on the right shows the corresponding elements of the generated results.

[1]Only the developer of the parallel Snap! environment needs to be aware of this; it is otherwise transparent to the end user.

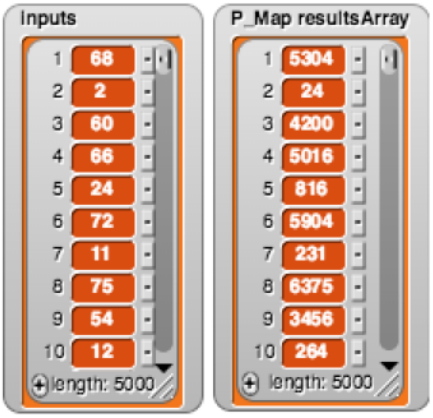

Fig. 14. Input and output lists for `ParallelMap`.

```
Process.prototype.reportParallelMap = function(aContext, aList, aCount) {
//Process.prototype.reportParallelOp = function(aContext, aList, aCount) {
    // At each runStep check if workers are done and return result

    // Use the context input array to store the parallel job:
    // [0] - ringified reporter obj
    // [1] - list
    // [2] - number of workers (default = #CPU's or 4)
    // ------------------------------------------------------
    // [3] - Parallel object

    var aFunction, anArray, body, p;

    body = 'return ' + aContext.expression.mappedCode() + ';';
    aFunction = new Function(aContext.inputs[0], body);

    if (this.context.inputs.length < 4) {
        if (aList instanceof List) {
            p = new Parallel(aList.asArray());
        } else {
            p = new Parallel(aList);
        }
        p.map(aFunction);
        this.context.inputs[3] = p;
    } else {
        p = this.context.inputs[3];
        if (p.operation._resolved) {
            return new List(p.data);
        };
    }
    this.pushContext('doYield');
    this.pushContext();
};
```

Fig. 15. Implementation code for `parallelMap`.

Figure 15 shows a listing of the code that implements the `parallelMap` block. The critical step is to perform the code mapping to create a dynamic function that can be passed to the `Parallel` object. If fewer Workers are created than there are list elements, the current solution is for Workers, as they become available, to process the next element in the list.

Our approach to adding explicit parallel constructs to Snap! utilizes important features of Snap!, such as *first-class lists*, *higher-order blocks*, and *code mapping*. We integrate HTML5 Web Workers with the Snap! implementation to allow true parallel execution of parallel tasks.

IV. Related Work

Snap! also includes a form of implicit parallelism through a feature that enables sprites to spawn *clones* of themselves that can operate independently from the parent sprite as well as from each other. Figure 16 shows the Snap! blocks associated with cloning as well as the result of running the program shown. We exploit this native cloning feature of Snap! to create yet another form of parallelism as explained below.

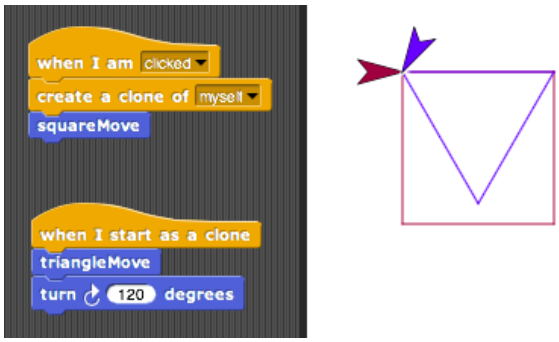

Fig. 16. Snap! `clone` blocks.

In related work, Figure 17 shows a parallel concession stand demo that features another native block we developed for the Snap! language called `parallelForEach`. Like the `parallelMap` reporter block discussed in the previous section, the `parallelForEach` block also operates over elements of an input list. In this case, however, instead of applying an operator to the list element, the list element is used as an input value to the script blocks nested inside the `parallelForEach` block.

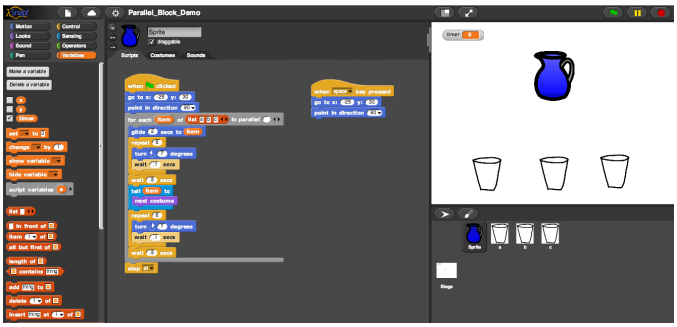

Fig. 17. Snap! interface showing a parallel concessions demo.

The `parallelForEach` block, shown in closer detail in Figure 18, operates in two different modes: parallel and sequential. In parallel mode, which is indicated by the label "in parallel" being visible, the system spawns clones of the Pitcher sprite to serve drinks to the waiting cups. The empty input box to the right of the "in parallel" label of the block allows the user to specify the level of parallelism, that is, the number of clones that will be spawned to execute the block. If empty, it defaults to the length of the input list.

During execution, each clone of the Pitcher sprite receives a copy of the nested script and a different element of the input list to use as input to the script, which in this case contains the names of the cup sprites that are awaiting beverage service.

Figures 19 through 21 show subsequent screen shots as the parallel version of the program progresses. The timer in the upper left shows the elapsed time.

In this parallel example of the Producer-Consumer paradigm, the program executes in three seconds using three concurrently executing clones of the Pitcher sprite that are spawned automatically when the block process is executed. The capability exemplified in this demo uses `Snap!`'s intrinsic cloning feature in a novel way to visually demonstrate parallel behavior.

By way of contrast, Figures 22 through 24 show what the result is in sequential mode. Without changing anything else, the user

Fig. 18. `ParallelForEach` block in parallel mode.

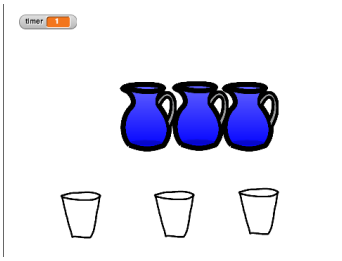

Fig. 19. Parallel demo at time-step 1.

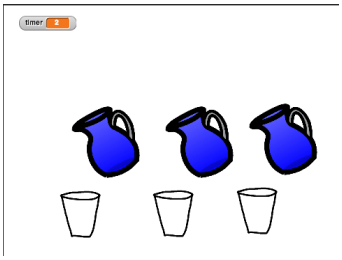

Fig. 20. Parallel demo at time-step 2.

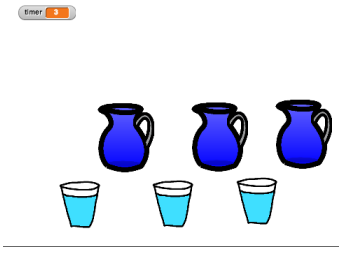

Fig. 21. Parallel demo at final time-step 3.

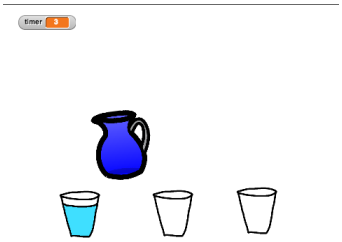

Fig. 22. Sequential demo at time-step 3.

can switch the `parallelForEach` block to sequential mode by collapsing the parallel input box, as shown in Figure 25. This tells the underlying system not to spawn clones and that the Pitcher sprite should execute the script as a normal `forEach` block by looping over the input array. In this case, the program takes 12 seconds to execute, versus the 3 seconds in parallel mode. The `parallelForEach` block provides a useful pedagogical tool for visually demonstrating the benefits of parallelism.

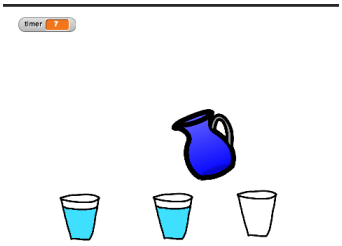

Fig. 23. Sequential demo at time-step 7.

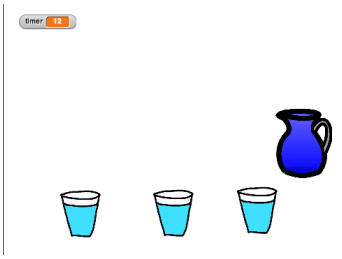

Fig. 24. Sequential demo at time-step 12.

V. Conclusions and Future Work

To address the pedagogical need for parallel computing, particularly in light of its ubiquity, our work seeks to augment the Snap! visual programming language with capabilities that would enable the execution of truly parallel processes. We have demonstrated a new Snap! block that executes in an explicitly parallel fashion. The implementation allows the user to dynamically specify an operation of any complexity that can subsequently be translated to the correct form for any designated back-end system. In the case scenario demonstrated in this paper, the back-end system is `Parallel.js`, which requires a mapping of Snap! blocks to JavaScript.

This same approach can be used to generate the back-end code for any target system, including those with more sophisticated architectures. This would provide a gateway for novice programmers to learn about and to utilize *explicit* parallel constructs at an earlier

Fig. 25. `ParallelForEach` block in sequential mode.

point in their programming careers than is currently the norm. It is our position that current parallel constructs in text-based languages such as C are not at a high enough level of abstraction to be accessible to the novice user and that such limitations are simply an artificial barrier that can be overcome through creative solutions. Our use of Snap! to implement `parallelMap` successfully demonstrates one such creative solution and paves the way for many more.

VI. Acknowledgement

The work was support in part by NSF ACI-1353786. The authors would also like to acknowledge Mark Gardner and Eli Tilevich for their insightful feedback on earlier versions of this work.

References

[1] The College Board, "College Board Officially Launches New AP Computer Science Principles Course to Increase Student Engagement in Computing," Dec 2014. [Online]. Available: https://www.collegeboard.org/college-board-officially-launches-new-ap-computer-science-principles-course

[2] ——, "College Board and NSF Expand Partnership to Bring Computer Science Classes to High Schools Across the U.S." June 2015. [Online]. Available: https://www.collegeboard.org/releases/2015/college-board-and-nsf-to-bring-computer-science-classes-to-high-schools

[3] E. Joseph and A. Snell and C. Willard, "Council on Competitiveness Study of U.S. Industrial HPC Users," July 2004. [Online]. Available: http://www.compete.org/pdf/HPCUsersSurvey.pdf

[4] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[5] B. Harvey, D. Garcia, J. Paley, and L. Segars, "Snap!:(build your own blocks)," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 2012, pp. 662–662.

[6] W3C, "Web workers, w3c working draft 24 september 2015." [Online]. Available: https://www.w3.org/TR/workers/

[7] L. Meadows and T. Mattson, "A hands-on introduction to openmp," November 2008.

[8] A. Silberschatz, P. Galvin, and G. Gagne, pp. 181–182.

[9] B. Harvey and J. Monig, "Snap! reference manual," p. 64. [Online]. Available: http://snap.berkeley.edu/SnapManual.pdf

[10] A. Savitzky and S. Mayr, "Parallel.js," accessed: 2015-11-16. [Online]. Available: http://adambom.github.com/parallel.js