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 a 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 smartphones. 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 currently 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 2004 study conducted by the U.S. Council of Competitiveness and sponsored by DARPA, 97% of surveyed U.S. businesses, including many Fortune 500 companies such as Procter & Gamble, noted that they could *not* exist or compete without the innovative use of high-performance

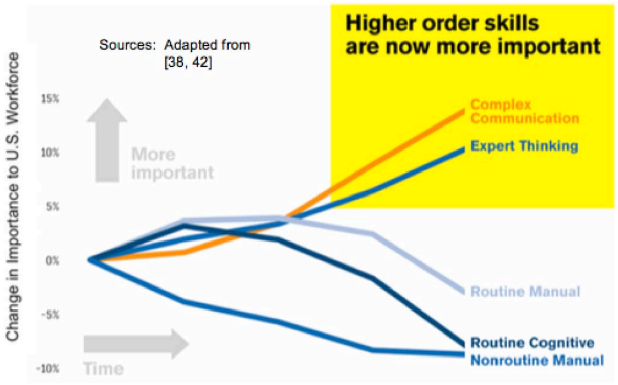

Fig. 1. Technological Changes Affecting U.S. Workforce Skills.

parallel computing (HPC) [3]. Unfortunately, those same companies lament the dearth of a trained workforce that is familiar with parallel computing concepts.

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 smartphones. 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 (serial) computing. We see the broad appeal of these environments due to the following two features. First, block-based languages possess 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 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.

978-1-5090-3682-0/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPSW.2016.194

To assist in addressing the need to improve the teaching of parallel computing concepts, we propose 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 serial 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 that we have done to demonstrate the viability of this approach. Finally, in Section V, we present our conclusions and future work.

II. Background

Scratch is a visual "drag-and-drop", blocks-based programming language developed at MIT. It introduces new programmers to a virtual world in which they control the actions and behaviors of characters called *Sprites*. Programmers assemble sequences of basic building blocks into more complex sets of behaviors that define movement and various interactions. Using the basic set of Scratch building blocks, programmers with very little computer science background can build seemingly complex games and animations with relative ease. Scratch is intended as a "gateway" language leading to more advanced computer programming using text-based languages. Scratch introduces novice users to fundamental concepts such as programming logic, variables, and control structures like branching with `if-then-else`, looping with `while` constructs, and input/output capabilities, among many other things.

Snap! is a visual programming language that is based directly on Scratch. Snap!'s look and feel is very similar to that of Scratch. However, Snap! deviates from Scratch in several important ways that make it ideal as a foundation upon which to launch our research.

First, Snap! is written in JavaScript so that it can run in a web browser, whereas Scratch was originally written in Smalltalk as a standalone application. While this alone is not a necessary feature, the fact that users can run the blocks-based environment in their web browsers without having to download a separate application makes it more attractive to people who might not otherwise use the software due to a fear of installing potentially harmful code onto their computers. (Note: Scratch has since been rewritten in Adobe Flash so that it, too, runs in a web browser.)

Second, Snap! introduces the notion of letting users define their own blocks using other blocks, something that Scratch does not enable. In fact, the original name for Snap! was *Build Your Own Blocks* (BYOB) to illustrate a rather significant way in which it differed from Scratch.

Third, Snap! treats lists as first-class objects, which again, is something that Scratch does not do. First-class lists, along with first-class procedures, which is what the block-building feature amounts to, makes Snap! a full-fledged programming language. Snap!, with its advanced features, loses none of its appeal to novice users who can safely ignore these advanced features when coding their applications. At the same time, Snap! gains the audience of experienced programmers who see a complete language that is suitable for solving their own programming needs.

Figure 2 shows what the Snap! interface looks like. The white area in the upper right of the interface is the stage where the sprites appear, exhibit their behavior, and display their output. In the center is the script editor, where the end user assembles the blocks that define the set of programs for the currently selected sprite. In the Snap! world, a project, or application, is built with one or more sprites,

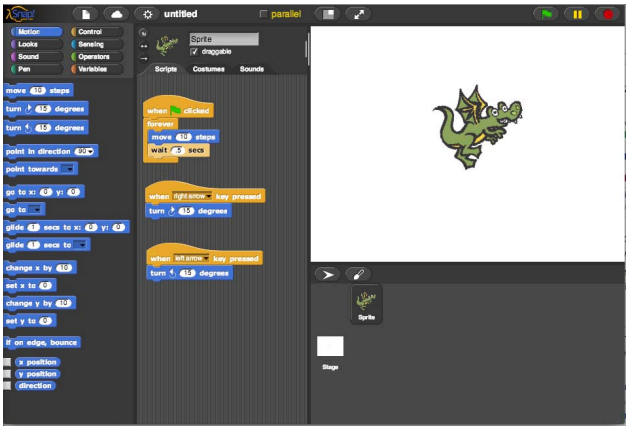

Fig. 2. The Snap! Graphical User Interface.

each having one or more scripts associated with it. Activated scripts run concurrently, both within a sprite's own collection of scripts and across all sprites.

Snap! adopts an event-driven programming model to build in interactivity into the system. Users can specify which events their sprites respond to and what the sprites do when they receive their specified events. For instance, the user clicking on the green start flag above the stage causes all the scripts beginning with the "when green flag clicked" block to start executing.

In the example shown in Figure 2, the project consists of a single dragon sprite having a program consisting of a collection of three separate scripts. The top script begins executing when the green flag button is pressed by the user. Because of the `forever` block, this script runs until the stop button is pressed. The middle script executes a single block of code telling the dragon to turn right every time the user presses the right arrow key. Similarly, the bottom script turns the dragon to the left every time the user presses the left arrow key. Figure 3 shows these scripts in detail.

Fig. 3. Dragon scripts.

Events are passed down to the run-time system at the heart of the Snap! programming environment, which identifies the active scripts and executes them in parallel. 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*. This form of parallelism is referred to as *concurrency*.

When events occur in the Snap! run-time environment, all scripts that wait for that event in order to execute 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 dragon scripts in Figure 2 results in the visual effect of the user seemingly being able to control the flight of the dragon.

This concurrent programming model exemplifies a form of *implicit* parallelism, in that the creation and execution of parallel processes is automatically managed by the Snap! run-time environment, along with access to shared and private data. It can be also be considered somewhat *explicit* by virtue of the user being able to write scripts independently for each sprite among a collection of sprites existing within the same project, as well as being able to write multiple separate scripts for a single sprite.

Programming in a visual language such as Snap! allows the user to break away from strict sequential coding by enabling the layout of separate block-sequences of scripts within a two-dimensional (2-D) visual editor, thus, identifying "multiple concurrent flows of control...naturally side-by-side" [6]. Therein lies the basis for using a visual language such as Snap! for trying to develop new abstractions for teaching parallel programming concepts: users are programming in parallel without necessarily being aware that that's what they are doing! Snap!'s execution model makes mimicking the real world seem natural and intuitive and provides an exciting and promising setting for our research.

If we assume that a visual programming language such as Snap! is the introductory programming language for a novice programmer, we can use this as an opportunity to "get in on the ground floor," so to speak. We aim to tap into this inherently parallel virtual world and take it to the next level by providing new constructs that will promote the kinds of logical thinking that is necessary when it comes to parallel programming. We aim to be able to teach parallel concepts at a point before people become too ingrained in a sequential way of thinking that could reasonably result from having learned to program using a text-based language that promotes a strictly linear way of 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 [7] seeks to address some of the problems that arise 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 4 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 5. Unlike 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 4, 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

Fig. 4. Snap!'s map block.

Fig. 5. Results of `map(x 10)`.

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 [8], [9], 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. In OpenMP, a master program executing sequentially can decide to execute faster by splitting a task amongst a number of workers which execute in parallel with respect to one another.

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 6 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. 6. `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 [10]. Figure 7 shows a small sampling of all the mapping constructs needed to translate Snap! blocks into JavaScript code.

Figure 8 shows a Snap! implementation of the `map` example from Figure 4. 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 9, according to an internal mapping that the 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.

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 10.

By leveraging this mapping to JavaScript capability, we can 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 [11]. Figure 11 shows

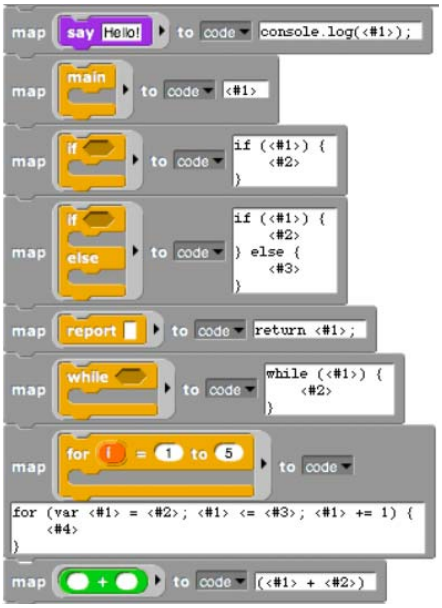

Fig. 7. Portion of Snap! to JavaScript code mapping.

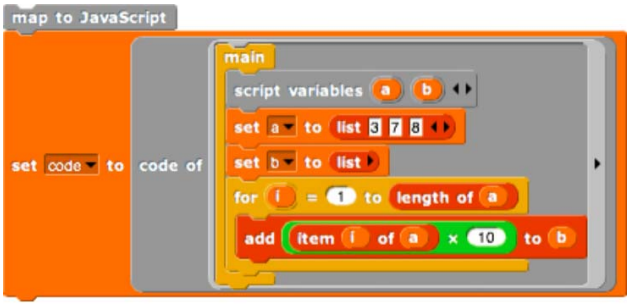

Fig. 8. Snap! script to map to JavaScript.

how easily the library can be used to create Web Workers.[1]

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

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

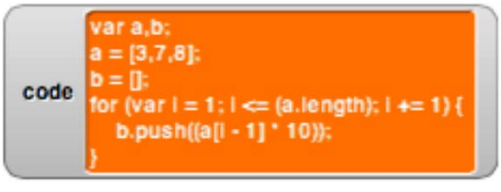

Fig. 9. Code mapping to JavaScript.

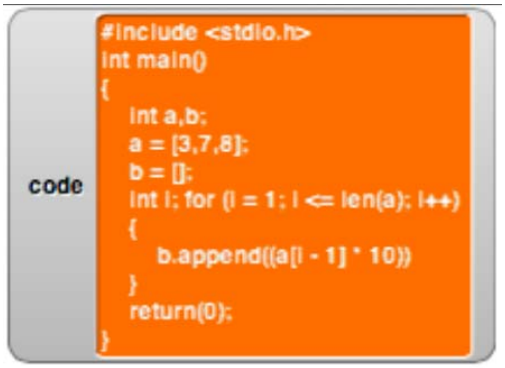

Fig. 10. Code mapping to C.

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

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

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

Fig. 11. Example code using Parallel.js.

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.

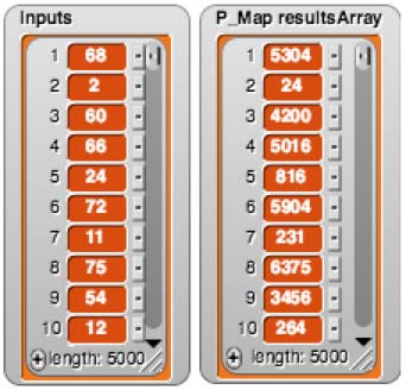

Fig. 12. Input and output lists for `ParallelMap`.

The new `parallelMap` block introduced in Figure 6 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 12 shows two lists corresponding to this example of the

```
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. 13. Implementation code for `parallelMap`.

`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.

Figure 13 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.

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 14 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.

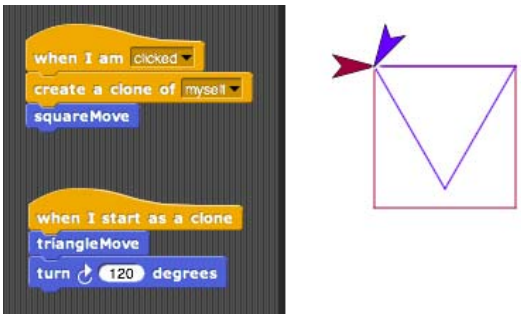

Fig. 14. Snap! `clone` blocks.

In related work, Figure 15 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.

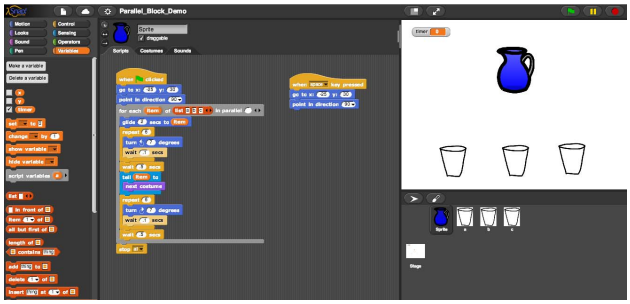

Fig. 15. Snap! interface showing a parallel concessions demo.

The `parallelForEach` block, shown in closer detail in Figure 16, 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.

Fig. 16. `ParallelForEach` block in parallel mode.

Figures 17 through 19 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.

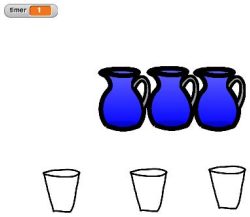

Fig. 17. Parallel demo at time-step 1.

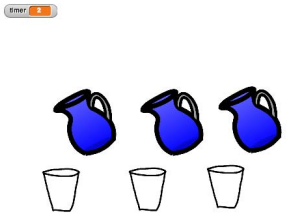

Fig. 18. Parallel demo at time-step 2.

By way of contrast, Figures 20 through 22 show what the result is in sequential mode. Without changing anything else, the user can switch the `parallelForEach` block to sequential mode by collapsing the parallel input box, as shown in Figure 23. 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.

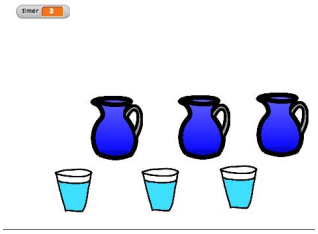

Fig. 19. Parallel demo at final time-step 3.

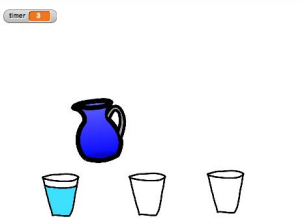

Fig. 20. Sequential demo at time-step 3.

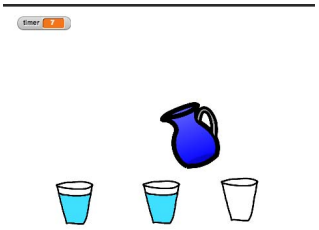

Fig. 21. Sequential demo at time-step 7.

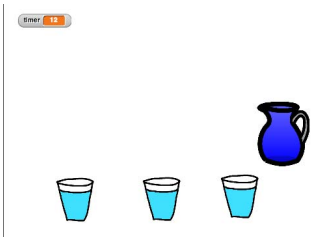

Fig. 22. Sequential demo at time-step 12.

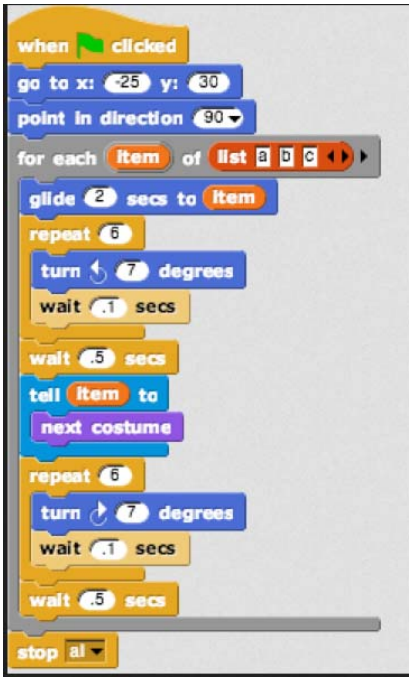

Fig. 23. `ParallelForEach` block in sequential mode.

IV. Related Work

Visual languages are seen as a way to make programming more accessible to novice users. They can reduce or eliminate the problems arising from incorrect syntax of a text-based program. In addition, the visual nature of these languages make it easier for programmers to understand the structure of their programs. The languages seek to provide high-level abstractions that hide unnecessary complexity, thus enabling the user to focus more on the logic of the programming task and less on the syntax of programming. Visual parallel programming languages more specifically seek ways in which to manage and ease the complexity of parallel programming that arises due to having to manage multiple threads of control.

Explicitly parallel programs are multi-dimensional objects; the natural representations of a parallel program are annotated directed graphs [12]. Indeed, a survey of the visual parallel programming scene shows a proliferation of languages modeled on the basis of data flow, control flow, and even object flow. LabVIEW is an example of a successful visual parallel programming language with the specific application of designing instrumentation [13].

In [14], the authors present an analysis of a gamut of visual programming languages, concluding that the success of a language is associated with specialization, e.g., LabVIEW. General-purpose visual programming languages for parallel programming require expertise, and subsequent visual diagrams become too cumbersome and no longer have an advantage over a text-based solution.

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 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] P. A. Lee and J. Webber, "Taxonomy for visual parallel programming languages," University of Newcastle upon Tyne, Computing Science, Tech. Rep., 2003. [Online]. Available: http://www.cs.ncl.ac.uk/publications/trs/papers/793.pdf

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

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

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

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

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

[12] J. C. Browne, J. Dongarra, S. I. Hyder, K. Moore, and P. Newton, "Visual programming and parallel computing," University of Texas Austin and University of Tennessee at Knoxville, Tech. Rep., 1996.

[13] [Online]. Available: http://www.ni.com/labview/

[14] V. Averbukh and M. Bakhterev, "The analysis of visual parallel programming languages," *Advances in Computer Science: an International Journal*, vol. 2, no. 4, pp. 126–31, 2013.