Parallel Programming with Pictures is a Snap!

Annette Feng[a,1], Mark Gardner[a,1,3], Wu-chun Feng[a,1,2,*]

[a] *Virginia Tech, Blacksburg, Virginia, U.S.A.*

Abstract

For decades, computing speeds seemingly doubled every 24 months by increasing the processor clock speed, thus giving software a "free ride" to better performance. This free ride, however, effectively ended by the mid-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 a major challenge: the difficulty in easily extracting parallel performance via current software abstractions. Consequently, this paper presents an approach that reduces the learning curve to parallel programming by introducing such concepts into a visual (but currently sequential) programming language called Snap!, which was inspired by MIT's Scratch project. Furthermore, our proposed visual abstractions can automatically generate parallel code for the end user to run in parallel on a variety of platforms from personal computing devices to supercomputers. Ultimately, this work seeks to increase parallel programming literacy so that users, whether novice or experienced, may leverage a world of ubiquitous parallelism to enhance productivity in all walks of life, including the sciences, engineering, commerce, and liberal arts.

Keywords: explicit parallel computing, computer science education, block-based programming, visual programming, parallel computational patterns, pedagogical tools, programming environments, languages for PDC and HPC

*Corresponding author

Email addresses: afeng@vt.edu (Annette Feng), mkg@vt.edu (Mark Gardner), wfeng@vt.edu (Wu-chun Feng)

[1]Department of Computer Science

[2]Department of Electrical and Computer Engineering

[3]Advanced Research Computing, Office of IT

1. 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 the National Science Foundation (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].

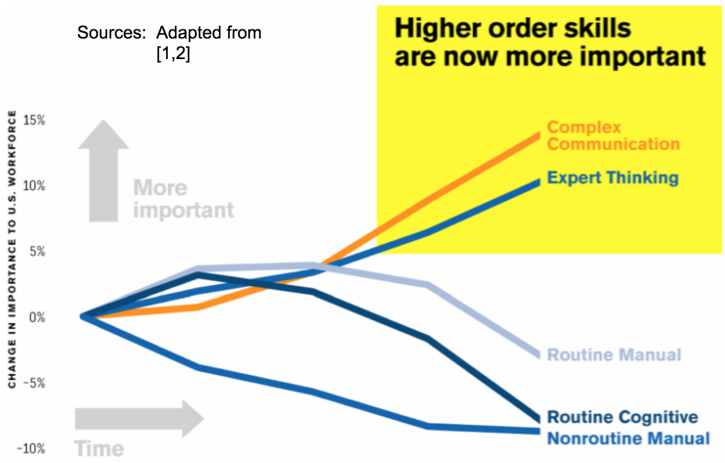


Figure 1: Technological Changes Affecting U.S. Workforce Skills.

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 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-2000s, computing capability seemingly doubled every

24 months by increasing the clock speed and giving software a "free ride" to better performance. However, 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 a major challenge: the difficulty in extracting parallel performance via current software abstractions. 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 (sequential) computing. We see the broad appeal of these environments due to the following two features. First, block-based languages possess a low barrier to entry. That is, students with no prior programming experience can quickly develop 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 in computing, while still providing enough basic blocks to provide a rewarding programming experience to novices. Because of these properties of block-based languages, we see them as fertile ground for introducing parallel computing concepts to a wide range of computing students — from K-12 students to college students to professionals.

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 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.

Further, we propose using an experimental feature of Snap!, one that produces a textual representation of the blocks in order to generate source code that can compile and run with high performance outside of the block-based environment, thus enabling the language to become an interactive development environment (IDE) that supports parallel programming for domain scientists (e.g., biologists and chemists) and domain engineers (e.g., aerospace engineers and mechanical engineers) as well. In this way, we believe individuals of all ages can become parallel programmers and utilize parallelism no matter their level of computing sophistication. By doing this, we believe that the workforce will be better prepared for the realities of our current parallel computing landscape.

To summarize, our goal is to increase parallel programming literacy for both the novice and the experienced. We aim to (1) introduce parallel programming concepts in a way that can be readily grasped by students as early as sixth grade

and (2) enable domain experts, who are typically not professional programmers, to employ parallelism to accomplish their work.

The rest of the paper is organized as follows. Section 2 covers background material, including a discussion of the block-based Snap! programming environment and various parallel paradigms. Section 3 presents our approach to teaching parallelism via the block-based Snap! programming environment and presents several examples of different kinds of parallelism. Section 4 discusses the implementation of the parallel constructs within our extended Snap! environment. Section 5 provides a brief report on a survey-based assessment of the deployment of our parallel programming environment during the "Women in Computing Day" (WCD) event at Virginia Tech on April 1, 2016. Section 6 extends the theme of "parallelism for the masses" by demonstrating how Snap! can be used to automatically generate text-based parallel code for subsequent compilation and execution. Specifically, our extension makes Snap! into an interactive development environment where programs written in the block language are translated to text-based parallel code for running outside of the browser on machines that range from smartphones to supercomputers. Finally, Section 7 discusses related work, and Section 8 summarizes the work.

2. Background

Scratch is a visual, drag-and-drop, block-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. 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. Using the basic set of Scratch building blocks, programmers with very little computer science background can build complex games and animations with relative ease. Scratch is intended as a "gateway" language leading to more advanced computer programming using text-based languages.

Snap! is a visual programming language that is based 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.[4] While this alone is not a necessary feature, the fact that users can run the block-based environment in their web browsers without having to download a separate application makes it more attractive to people who might not otherwise

[4]Since beginning our research, Scratch has been re-written in Adobe Flash so that it, too, can run in a web browser.

use the software due to a fear of installing potentially harmful code onto their computers.

Second, Snap! allows users to define their own blocks using other blocks, something that Scratch does not support. 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 support. 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 achieving their own programming needs.

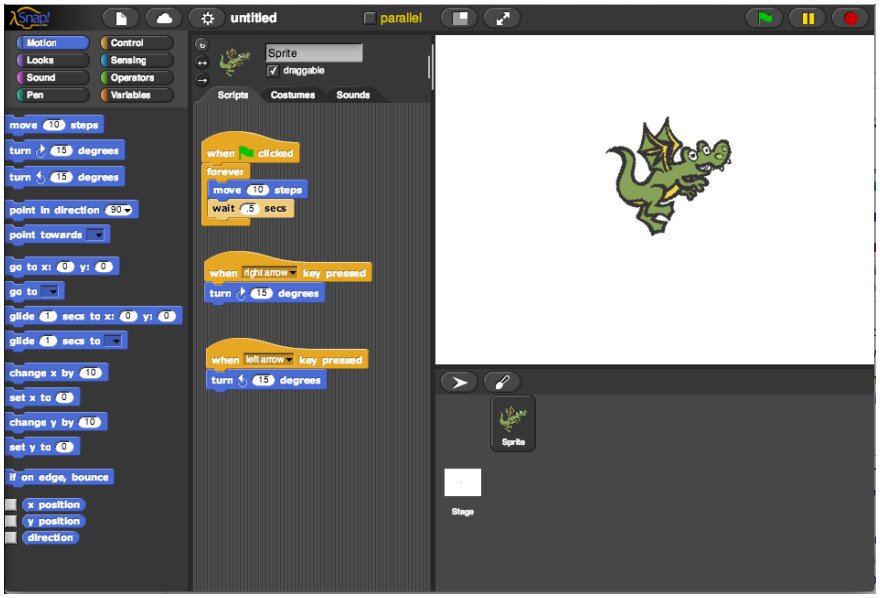


Figure 2: The Snap! Graphical User Interface.

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 (or programmer) 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, 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 incorporate 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 that begin 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 execution when the green flag button is pressed by the user. Because of the `forever` block, this script runs "forever" 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.

Figure 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 effectively 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 delivers 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 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 to teach 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 are necessary when it comes to parallel programming. We aim 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.

3. Introducing Parallelism into Snap!

Performing operations across a collection of items is an approachable way to introduce parallelism. Snap! includes a `map` block that (sequentially) executes the specified operation on each element of a list and returns a list of the results. The mapping of an operation across the elements of the list *in parallel* is a logical and easily grasped extension to the sequential Snap!

3.1. The Map Block

Figure 4a shows the existing 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 4b. 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.

(a) Multiply each list element by 10. (b) Resulting list.

Figure 4: Snap!'s Map Block.

The `map` function executes sequentially by looping over a list, applying the user-supplied function to each list element, and ultimately, returning a new list

containing the results. Upon closer inspection of the `map` block of Figure 4a, 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.

3.2. The Parallel Map Block

Figure 5 introduces our new `parallelMap` block. The new block integrates the visual representation of the original `map` block with a back-end implementation that utilizes explicitly parallel HTML5 Web Workers in the place of Snap!'s sequential execution model. The new `parallelMap` block also has an optional 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 HTML5 Web Workers to spawn. By default, four Web Workers are created.

Figure 5: `parallelMap` Function.

Figure 6 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 `parallelMap` block in the example, and the one on the right shows the corresponding elements of the generated results. The implementation of the parallel map block is given in Section 4.

3.3. The Parallel ForEach Block

The next parallel construct we introduce is a `parallelForEach` block. Like the `parallelMap` 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 each list element and returning a list, each list element is used as an input value to the script blocks nested inside the `parallelForEach` block with no result returned.

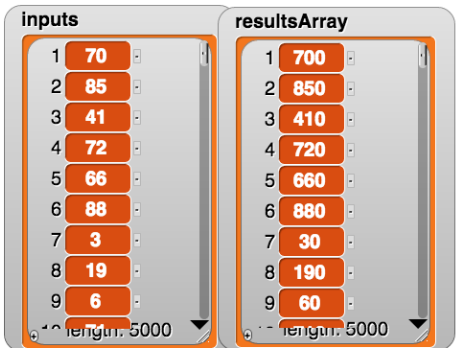


Figure 6: Input and Output Lists for `parallelMap`.

We created a graphical illustration which uses the `parallelForEach`, along with moving Snap!'s sprites on the stage, to give an intuitive understanding of how much parallelism speeds up operations in a way that is easily understood by those not familiar with parallel programming. The example simulates a concession stand and can be run in either sequential or parallel mode (see Figure 7). A timer in the upper left corner shows the elapsed time in Snap! timestep units. In sequential mode, a single pitcher pours the three drinks one at a time for a total of twelve timesteps. In the parallel mode, three pitchers pour a single drink simultaneously for a total of only three timesteps.[5]

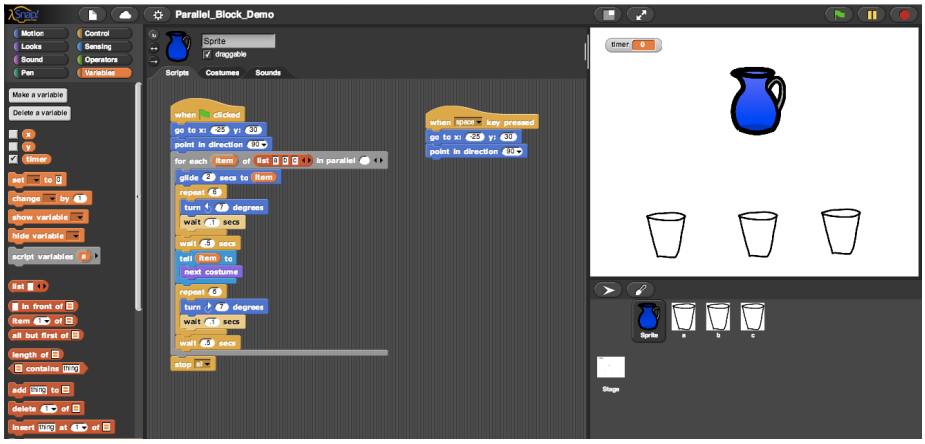
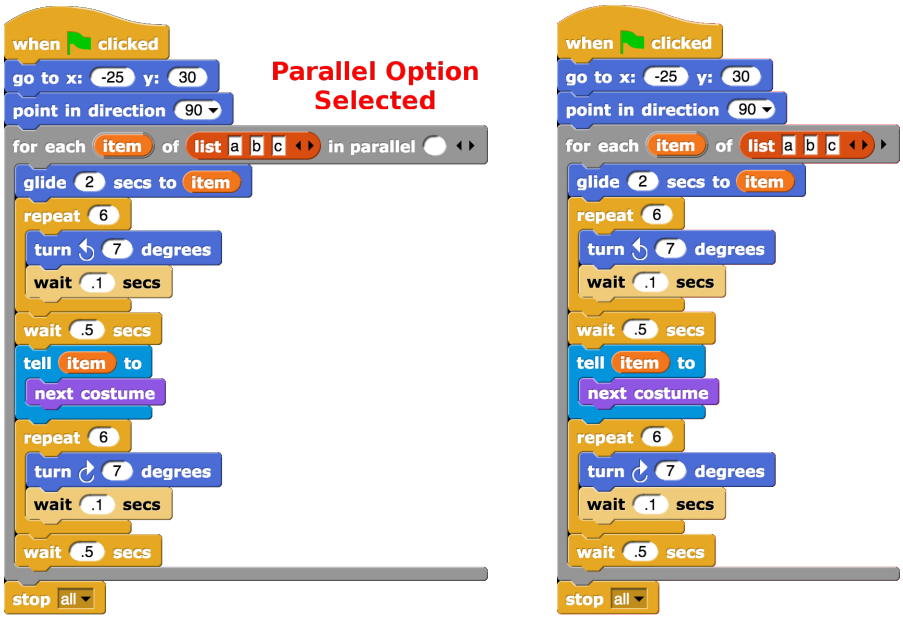


Figure 7: Snap! Interface Showing a Parallel Concession Stand Example.

[5]It takes three timesteps to fill a glass so we would expect the sequential case to take nine timesteps rather than twelve. The unexpected difference is due to interference by other tasks that also execute in the browser or on the computer. As the sequential case takes longer to execute, the effect is more noticeable for it than for the parallel case. In this case, the difference happened to be three timesteps.

To enable comparison, our `parallelForEach` block can operate in two different modes: parallel (Figure 8a) and sequential (Figure 8b). 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, i.e., the number of clones that will be spawned to execute the block. If empty, it defaults to the length of the input list, which, in this case, is three.



(a) Parallel Mode.

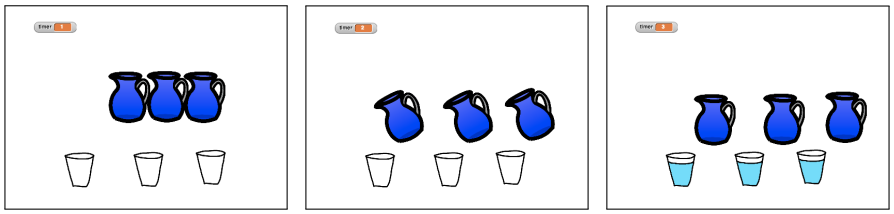
(b) Sequential Mode.

Figure 8: `parallelForEach` Block in Parallel and Sequential Modes.

During execution, each clone of the *Pitcher* sprite executes the same nested script on a different element of the input list, which, in this case, contains the name of the *Cup* sprite that awaits beverage service. Figures 9a through 9c 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 example uses Snap!'s intrinsic cloning feature in a novel way to visually demonstrate parallel behavior.

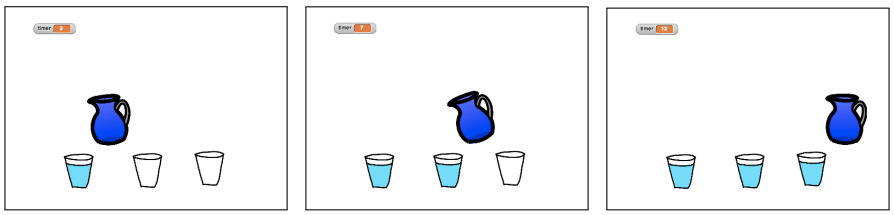
By way of contrast, Figures 10a through 10c show the result in sequential mode. Without changing anything else, the user can switch the `parallelForEach` block to sequential mode by collapsing the parallel input box. This tells the underlying system not to spawn clones and that the *Pitcher* sprite should



(a) Timestep 1. (b) Timestep 2. (c) Timestep 3 (final).

Figure 9: Parallel Example over Time.

execute the script as a normal `forEach` block by looping over the input array. In this case, the program executes in 12 seconds. The `parallelForEach` block provides a useful pedagogical tool for visually demonstrating the benefits of parallelism.



(a) Timestep 3. (b) Timestep 7. (c) Timestep 12.

Figure 10: Sequential Example at Timestep 12.

3.4. The MapReduce Block

The rise of "big data" has led to increased need to process large amounts of data. The MapReduce paradigm [7, 8] has emerged as one of the most popular techniques for processing such data. In order to teach this important parallel computing approach, we provide a MapReduce block for Snap!.

Figure 11 shows the Snap! blocks for a canonical MapReduce example, namely *word count*. In a `MapReduce` block, the first argument is a block that implements the *map* function. The map function is executed for each item in the supplied list, mapping the item to a value. The function returns a two-element list with the item as the *key* and the result as the *value*. The result of mapping the function over all the elements of the input list is also a list. Many invocations of the map function can operate on separate items in parallel because the map function only depends upon a single item, its input, from the list. This inherent lack of dependencies between items is what allows for convenient expression of a parallel computation.

The second argument to the MapReduce block is also a block, one which implements a *reduce* function. Just like the map function, the reduce function

Figure 11: Word Count Implemented in Snap!.

operates upon each item in the intermediate list from the map phase but, unlike the map function, the computation it performs may depend upon previous items in order to perform the reduction operation.[6] It too executes in parallel. The final argument to the MapReduce block is the input list, in this case, generated from a string of words.

In the MapReduce paradigm, the map or reduce functions can express the identity function which passes its input argument through unchanged. Clearly such operations are redundant but because they are executed in parallel in MapReduce, their execution often occurs at the same time as meaningful operations and hence do not contribute much if any to the overall execution time. The convenience of having a uniform model of parallel computation outweighs the bit of extra overhead that comes from executing the identity function.

The result of the *word count* example, as shown in Figure 12, is a sorted list of unique words from the input with the number of times the words appear.

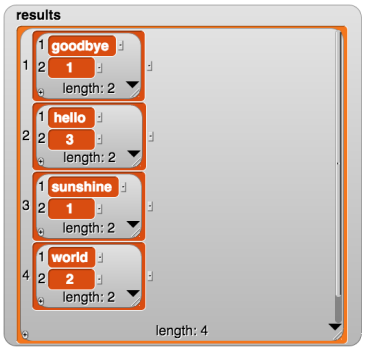


Figure 12: Output of Word Count Implemented in Snap!.

Although conceptually simple, MapReduce implementations can be quite complex to set up and use. Fortunately, these details are hidden in the implementation of the MapReduce block in Snap!.

We use global climate modeling as a real-world example of MapReduce that is suitable for an earth science class. Utilizing weather station data from the National Ocean and Atmospheric Administration (NOAA), which contain tem-

[6]The elements of the intermediate result are sorted by the value of the key in between the map function and the reduce function, as required by the semantics of MapReduce.

peratures in Fahrenheit, students can convert the temperatures to Celsius and compute their average. By asking them to observe the average temperature over many years using the NOAA data, students can attempt to observe a mean change in the temperature of the Earth over time.

The temperature conversion from Fahrenheit to Celsius is performed in the map function shown in Figure 13. The reduce function computes an average over the converted values. Thus the effect of the MapReduce program is to take a list of Fahrenheit temperatures and produce an average temperature in Celsius as is required for this example.

Figure 13: A MapReduce Block for Snap!.

While it is unlikely that the computed averages will exactly reproduce the careful measurements and calculations by climate scientists, if for no other reason than climate is a global phenomenon while the data from NOAA is for the United States only, the exercise provides an opportunity for the middle-school teacher to use computing to tie into the Earth and Social Sciences. This leads into potential discussions about the social and political issues surrounding climate change and empowers the students to get involved. The somewhat abstract concepts of temperature conversion, climate vs. weather, government policy, and social responsibility become more concrete and meaningful, all while teaching parallel programming concepts.

4. Implementation

Up to this point, we have focused on using Snap! with parallel extensions to teach parallel programming concepts. We now describe the underlying implementation of the parallel extensions. These underlying implementation details are hidden from the end user (i.e., abstracted away) so that end users do not have to be bogged down by the syntactic tedium (or complexity) of a text-based language when creating and managing explicit parallel threads.

4.1. HTML5 Web Workers

Because the entire execution of Snap! scripts occurs within a single browser thread, computationally expensive scripts can slow down the execution of Snap! programs and even render the browser unresponsive. Furthermore, the best that can be done with a single browser thread is to simulate parallel execution by periodically switching between the execution of several processes within the browser thread in order to give the illusion of multiple processes executing at once, i.e., *concurrency*.

To achieve true parallel execution, we make use of HTML5 Web Workers [9]. HTML5 Web Workers provide a method for JavaScript programs to spawn separate background threads that can utilize the underlying multi-core architecture

of a host system, heretofore ignored by JavaScript programs. Each HTML5 Web Worker corresponds to a single thread and runs independently from other workers and independently from the user-interface thread, thus keeping the browser responsive to user input. HTML5 Web Workers are meant to process computationally-intensive calculations that would otherwise potentially render the browser unresponsive and are hence an ideal construct for introducing true parallelism into Snap!.

To enable access to HTML5 Web Workers within Snap!, we utilize Parallel.js [10] — a small open-source JavaScript library that can be integrated into any JavaScript project simply by loading it in the project's .html file. The code in Listing 1 shows a simple example that demonstrates the ease with which HTML5 Web Workers can be employed. The example maps a function which doubles the input value in parallel across all the elements in the input list using two HTML5 Web Workers. In spite of this simplicity, only the developer of the parallel Snap! environment needs to be aware of the details; it is otherwise transparent to the end user.

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

var p = new Parallel([1,2,3,4], {maxWorkers: 2});
p.map(mydouble);
console.log(p.data);
```

Listing 1: Example Code using Parallel.js.

We now 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.

4.2. Implementing Parallelism with HTML5 Web Workers

Listing 2 shows a listing of the code that implements the `parallelMap` block. It is an example of how HTML Web Workers are used to implement true parallelism in Snap!.

The code takes advantage of the fact that JavaScript can create functions on the fly. The mathematical operator specified by the user in the `parallelMap` block is extracted from the current stack frame, `aContext`. It is wrapped into a dynamically-created JavaScript function and assigned to variable `aFunction`, which is later passed as an argument to the parallel map: `p.map(aFunction)`. The list of values to be operated upon, `aList`, is passed as an argument during `Parallel` object creation, along with the number of HTML5 Web Workers to instantiate. If fewer workers are created than there are list elements, the workers systematically process the remaining elements from the list until completed.[7]

[7]The `if` statement is an artifact of the way that Snap! implements concurrency as coroutines. The first branch of the `if` creates the HTML5 Web Workers then the `pushContext('doYield')` instructs the environment to allow something else to run. (The

```
Process.prototype.reportParallelMap =
                        function(aContext, aList, aCount) {
  // At each runStep check to see if the workers are done.
  // If so, return the resulting array and quit.
  //
  // 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, workers, p;
  if (this.context.inputs.length < 4) {
    body = 'return ' + aContext.expression.mappedCode() + ';';
    aFunction = new Function(aContext.inputs[0], body);
    workers = aCount || navigator.hardwareConcurrency || 4;
    if (aList instanceof List) {
      p = new Parallel(aList.asArray(), {maxWorkers: workers});
    } else {
      p = new Parallel(aList, {maxWorkers: workers});
    }
    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();
};
```

Listing 2: Implementation of `parallelMap`.

5. Assessment

We deployed and assessed our parallel Snap! environment at the 18th Annual *Women in Computing Day (WCD)* event, sponsored by the Association

---

second `pushContext()` pushes an empty context to prevent the `reportParallelMap` function from being popped by the caller. This leaves the process on the execution stack to be called again.) The second branch which is taken in subsequent calls to `reportParallelMap` checks to see if the HTML5 Web Workers are finished and returns the results.

of Women in Computing (AWC) and held at Virginia Tech on April 1, 2016.[8] The AWC is a student-run, non-profit organization at Virginia Tech, consisting of approximately 30 undergraduate students and 10 graduate students. Every year, the AWC holds a WCD event that hosts approximately 100 seventh-grade girls from 5-6 local middle schools, both rural and urban, for a day of fun with computing.

This year, the AWC partitioned the middle school girls into four groups of 24-25 in order to have each group cycle through four parallel 50-minute activity sessions, of which our parallel Snap! was the only activity that involved any programming, as shown in Figure 14. Thus, every 50 minutes, our task entailed teaching a new set of 24-25 girls how to program and then how to do so in parallel! The informal curriculum first focused on the original (sequential) Snap! environment and gave the middle schoolers time to get accustomed to the (sequential) programming of the concession stand and global climate modeling examples (even though many had no idea that they were actually programming). Approximately 20 minutes through the time period, we then introduced parallelism via the `parallelMap` amd `parallelForEach` blocks. The students were then allowed to program on their own for the remainder of the session. One of the more creative examples of parallelism was a video game, where the player controlled an on-screen (laundry) basket and tried to catch water balloons that were falling from the sky (in parallel) before they landed on the heads of people.

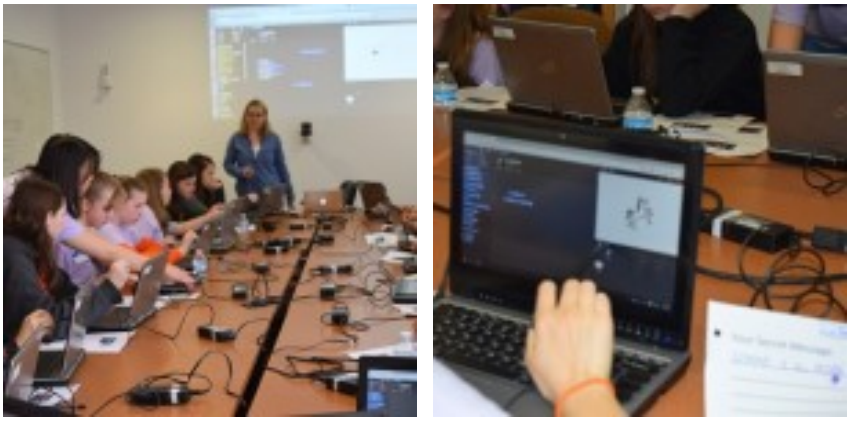


Figure 14: Women in Computing Day at Virginia Tech

While this parallel Snap! environment is intended for all users — whether K-12 students, undergraduate students, graduate students, or even professionals

[8] While this programming environment was deployed and assessed with middle-school girls, it is important to note that this environment is intended to be accessible to both novice and experienced users, including undergraduate and graduate students as well as professionals with a desire to learn how to program in parallel.

— we deployed and assessed the curriculum with respect to middle schoolers. As such, relative to the CSTA K-12 Computer Science Standards on teaching computer science to K-12, we taught "computer science for problem solving."

Of the three levels of computer science learning (based on on CSTA's guidelines), we explicitly addressed the latter two levels, namely "Level 2 (recommended for grades 6-9) Computer Science and Community" and "Level 3 (recommended for grades 9-12) Applying concepts and creating real-world solutions." Of the five *strands* that orthogonally permeate the aforementioned levels of learning of computer science — (1) computational thinking; (2) collaboration; (3) computing practice; (4) computers and communication devices; and (5) community, global, and ethical impacts — we explicitly addressed the first three, i.e., computational thinking, collaboration, and computing practice.

Finally, our assessment during the 18th WCD event also included a brief survey that solicited brief written feedback from the participating middle schoolers. The survey asked the students to address (1) whether computer science would be a potential career choice for them after having gone through our specific WCD activity on parallel Snap!; (2) whether they thought that their career choice (if not computer science) could benefit from computer science; and (3) whether their impression of computer science was more favorable, less favorable, or the same as before going through our WCD activity. From this sample of middle schoolers, 29% indicated computer science as a potential career choice, 54% indicated something other than computer science, and 17% provided no answer (or said that they did not know). Of those who indicated computer science as *not* being a potential career choice, 57% indicated that computer science would benefit their chosen career. Lastly, when asked if their impression of computer science was more favorable, less favorable, or the same as *before* being exposed to our parallel Snap! activity, 86% indicated more favorable, 9% indicated less favorable, and 6% indicated the same or had no opinion.

Self-Assessment: While WCD provided us with the opportunity and flexibility to teach parallel programming "how we wanted to" in a controlled setting, each teaching session was limited to only 50 minutes. As a consequence, we did not have sufficient time to more formally deploy and assess our parallel computing curriculum, including, for example, a comparison and contrast between sequential and parallel Snap! or between parallel Snap! and a text-based parallel programming language with respect to performance and programmability, i.e., ease of use.

However, due in part to the success of our parallel computing environment and curriculum at WCD, we have been invited to incorporate our parallel Snap! into a "CS0 Computer Literacy" course [11] on problem solving for the spring semester of 2017. The proposed one-week curriculum has been set up to assist first-year undergraduate students in developing problem-solving skills and applying these skills to both sequential and parallel programming activities, in accordance with many of the core curriculum goals found in the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing [11].

6. Parallel Snap! in the Real World

The execution environment for parallel Snap! so far has been the browser, which is a convenient way for individuals to become comfortable with the concepts. But clearly the world of parallel computing spans a greater breadth of environments. In this section, we discuss how programs written in Snap! with our parallel extensions are run in more traditional environments on large volumes of data instead of being restricted to running in a browser. We accomplish this by translating Snap! blocks into OpenMP source code which is compiled and executed in the manner of traditional parallel computing environments. With this approach, Snap! moves from being a pedagogical tool into an environment that supports serious parallel programming. We begin by providing some background on OpenMP and how Snap! blocks are translated into a textual form.

6.1. OpenMP

Our approach to introducing true parallelism to Snap! draws inspiration from OpenMP [12, 13], an application programming interface (API) that supports multi-platform, shared-memory, multiprocessing across a multitude of programming languages (e.g., C, C++, and Fortran), operating systems, and processor architectures. It is widely used in parallelizing scientific computations. In OpenMP, a master program executing sequentially can decide to execute faster by splitting a task among 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 for introducing parallelism into a sequential program. Listing 3 shows a simple sequential C program that prints out "hello(0), world(0)":

```
void main() {
  int ID = 0;
  printf(" hello(%d), ", ID);
  printf(" world(%d) \n", ID);
}
```

Listing 3: Sequential Program in C.

By adding a simple directive (or pragma) and a function call to obtain the thread ID, the previous example readily compiles into a parallel program, where each thread prints out its own "hello(*ID*), world(*ID*)" message, where *ID* is the thread ID. The parallel version of the sequential program is in Listing 4.

OpenMP is attractive because the difference between the sequential C version and the parallel OpenMP C version is very small and easily understood. This is in stark contrast to the complexity of other text-based approaches, such as pthreads.

```c
#include "omp.h"
void main() {
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf(" hello(%d), ", ID);
    printf(" world(%d) \n", ID);
  }
}
```

Listing 4: Parallel Program in OpenMP C.

In designing our parallel extensions to Snap!, we seek to emulate the simplicity of text-based OpenMP parallelism using the pragma approach, but with a block-based approach instead. As discussed previously, our research extensions to Snap! introduce new built-in blocks to support parallelism in the browser. To allow running programs created using Snap! blocks outside the browser, we leverage Snap!'s built-in *code-mapping* feature to generate text-based OpenMP code, which is then compiled and run on the target computing system.

6.2. Snap!'s Code Mapping Support

Snap! has an experimental feature that translates the visual, block-based programs of Snap! into any text-based programming language [14]. Through the use of this feature, parallel programs in Snap! are translated to OpenMP code ready to compile and run in traditional parallel computing environments.

Figure 15 shows a sampling of key code mapping constructs we use to translate Snap! blocks into C. Text of the placeholders `<#1>`, `<#2>`... signify the mapping of the first location in the block to be filled in, the second, and so forth. The remainder of the characters are copied to the output verbatim. Because Snap! programs consist of nested blocks, the value substituted for a particular placeholder may itself have resulted from the translation of a nested block.

Figure 16 shows a Snap! implementation of the non-parallel `map` example from Figure 4a. (In this example, the map operation is written out explicitly so that the code translation is easier to follow.)

The Snap! ‘‘`code of`’’ block, when executed, automatically translates the script it contains into the corresponding code for the language specified. Listing 5, shows C code according to the language mapping as specified by the ‘‘`map to C`’’ block at the top of the script in Figure 16. (The ‘‘`map to C`’’ block must be executed first to set the internal code mapping to the C programming language so that the subsequent execution of the ‘‘`code of`’’ block does the code translation correctly. The ‘‘`map to C`’’ block is elided in subsequent figures.) To change the back-end language to which the Snap! scripts are being mapped, i.e., if the user wishes to switch from C to JavaScript, the ‘‘`map to C`’’ block is changed to a ‘‘`map to JavaScript`’’ block to specify the appropriate mappings.

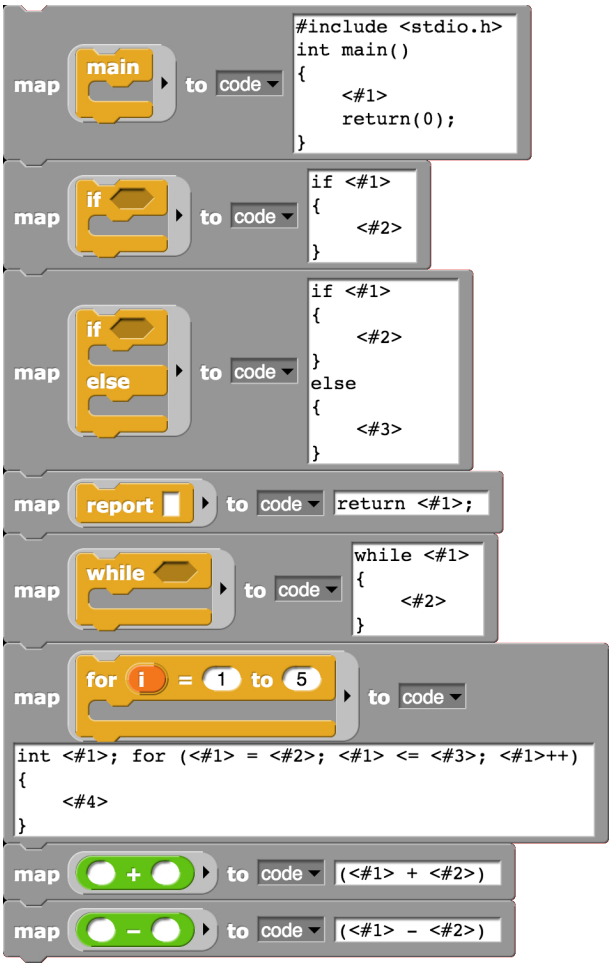


Figure 15: Portion of Snap! to C Code Mapping.

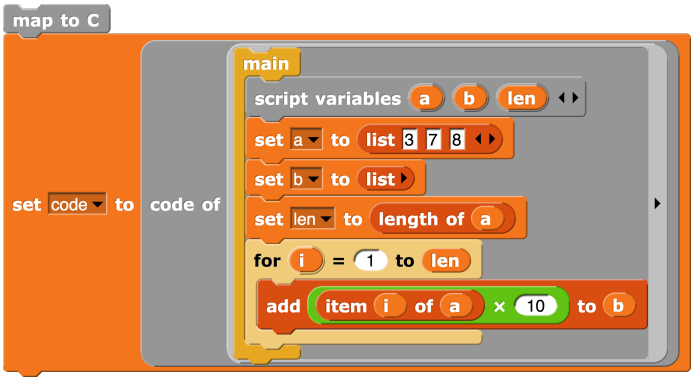


Figure 16: Snap! Script to Map to C.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
    int data;
    struct node *next;
} node_t;

void append(int d, node_t *p) {
    while (p->next != NULL)
        p = p->next;
    p->next = (node_t *) malloc(sizeof(node_t));
    p = p->next;
    p->data = d;
    p->next = NULL;
}

int main()
{
    int len;
    int a[] = {3, 7, 8};
    node_t *b = (node_t *) malloc(sizeof(node_t));
    len = (sizeof(a)/sizeof(a[0]));
    int i; for (i = 1; i <= len; i++)
    {
        append((a[i - 1] * 10), b);
    }
    return (0);
}
```

Listing 5: Code Mapping to C.

Currently, mappings exist for JavaScript, C, Smalltalk, and Python. Code mappings for new textual languages can easily be specified by the user by creating the corresponding mapping block.

6.2.1. Mapping Blocks to OpenMP

Not only are our extensions to Snap! modeled after OpenMP, but using the experimental code mapping feature blocks we are now ready to translate Snap! blocks into OpenMP code for compilation and execution in parallel outside of the browser.

The strategy is to generate a text file with functions containing OpenMP pragma annotations for parallel processing. The text file is then compiled and linked against an OpenMP run time to produce a parallel program. This workflow is shown in Figure 17. With this capability, Snap! is able to break free from being strictly a teaching language and begins to be a language and environment

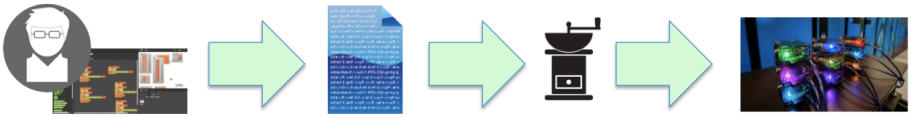


Figure 17: Snap! as part of a scientific workflow.

for creating programs of general utility.

To be concrete, consider the MapReduce application which converts Fahrenheit to Celsius and produces an average temperature introduced in Section 3.4. Figure 18 shows the map-reduce block and its corresponding mapped output code. Note that the textual output contains three placeholder symbols `<#1>`, `<#2>`, and `<#3>` corresponding to the three arguments it requires. The first placeholder symbol `<#1>` is replaced by the text from the block supplied as the mapping function. The substituted text is the body of the map function. The second placeholder symbol `<#2>` is replaced by the text from the block supplied as the reducing function. The substituted text is the body of the reduce function. The final placeholder is replaced by the input list.[9] Note: the `strncpy` function calls copy the keys from the input to the output per MapReduce semantics. The transformation of the values is performed by the bodies of the map and reduce functions as discussed above.

Figure 19 shows the code block for the the mapper which converts Fahrenheit to Celsius, along with the textual code produced for the mapper function by the code mapping. Likewise, Figure 20 shows the code block for the reducer which computes the average of all of the values given to it, along with it's mapped output code.

Listing 6 shows the combined map and reduce functions produced by performing the code mapping of the Snap! blocks to OpenMP. (For completeness, the file containing the main function is shown in Listing 7.) Compared with the Snap! block implementation it was generated from, the textual OpenMP representation is full of details which are necessary for successful compilation of the textual language but which are extraneous for a high-level description of the problem being solved. Those details are provided in the mapping from map-reduce to OpenMP code by the programmer implementing the map-reduce block, i.e., us. Thereafter, it is used without care by others for whom the intricacies of OpenMP coding is but a diversion from their work.

While the example was purposefully chosen to be small so the resulting translated OpenMP code did not consume too much space and yet represents a complete example, it demonstrates that Snap!'s experimental code mapping functionality is useful for generating OpenMP code which is then compiled and executed outside of the Snap! environment. In other words, Snap! is on its way to become an interactive development environment (IDE) for developing parallel

[9]The symbol `<#3>` signifies the list of data to be operated upon and is used in the driver code which invokes the map and reduce functions on each element of the list.

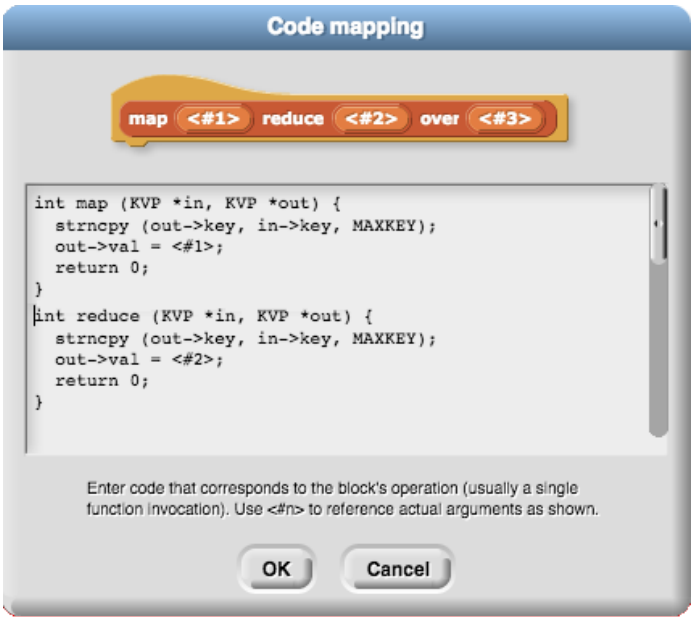


Figure 18: The MapReduce Block and Corresponding Mapped Code.

Figure 19: The Fahrenheit-to-Celsius Mapper Block and Corresponding Mapped Code.

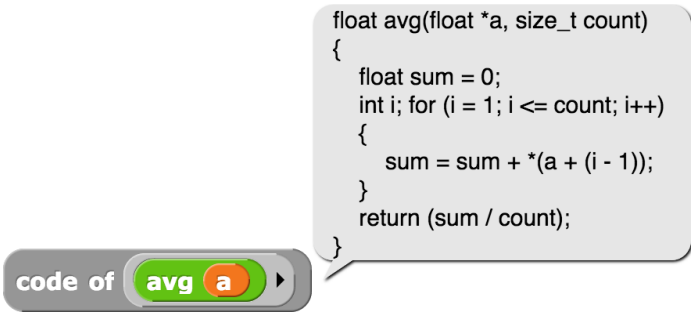


Figure 20: The Averaging Reducer Block and Corresponding Mapped Code.

```c
#include <math.h>
#include <string.h>
#include "kvp.h"

float avg(float *a, size_t count) {
  if (count == 1)
    return *a;
  return (*a + ((count-1)*avg(a+1,count-1))/count);
}

int map (KVP *in, KVP *out) {
  strncpy (out->key, in->key, MAXKEY);
  out->val = ((5 * (in->val - 32)) / 9);
  return 0;
}

int reduce (KVP *in, KVP *out) {
  strncpy (out->key, in->key, MAXKEY);
  out->val = avg(in->val);
  return 0;
}
```

Listing 6: Fahrenheit-to-Celsius Map and Reduce Functions After Mapping.

applications that can run in HPC environments.

6.3. Future Work

As discussed earlier, we envision that the Snap! environment could become an integrated development environment (IDE) for creating parallel programs for use by scientists of all domains. In this section, we consider what remains to be done for this vision to become a reality.

For use as an IDE, the Snap! environment needs a way in which to ingest larger amounts of data without having to enter them in one by one into a list box. For production use, it needs to have a way to consume existing data files. Likewise, it needs a way to write data to files for use by other programs outside of Snap!, and it needs to do these things in a consistent manner without compromising the user-friendly interface that it currently has.

Besides support for the mapping of blocks to OpenMP shown above and which we have already implemented, the Snap! environment needs to incorporate the means for automating the compilation and linking of the textual output from the code mapping process in order to fulfill the same requirements as are currently filled by the Makefile in command-line programming environments or by the project in graphical environments such as Eclipse.

Once the Snap! blocks are codified and compiled, the program needs to be scheduled for execution. In a workstation environment, the execution occurs on the local machine so scheduling is straightforward. Supercomputers, however,

```c
/* OpenMP driver for Parallel Snap! MapReduce code output. */
#include <omp.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "kvp.h"

int main(int argc, char *argv[]) {
  int nkvp;
  KVP *inputlist, *midlist, *outputlist;

  if (input(&nkvp, &inputlist) != 0) {
    return 1;
  }
  midlist = malloc(nkvp * sizeof(struct KVP));

  /* Run mapper */
  #pragma omp parallel for shared(nkvp, inputlist, midlist)
  for (KVP *i = inputlist, *m = midlist;
         i < inputlist + nkvp; i++, m++) {
    map(i, m);
  }

  /* Sort on keys */
  qsort(midlist, nkvp, sizeof(KVP), compare);
  outputlist = malloc(nkvp * sizeof(struct KVP));

  /* Run reducer */
  #pragma omp parallel for shared(nkvp, midlist, outputlist)
  for (int i = 0; i < nkvp; i++) {
    reduce(&midlist[i], &outputlist[i]);
  }

  if (output(nkvp, outputlist) != 0) {
    exit(1);
  }

  free(inputlist);
  free(outputlist);

  return 0;
}
```

Listing 7: MapReduce OpenMP Driver Code Containing `main`.

execute large, long-running jobs and use sophisticated batch scheduling systems. The Snap! environment can be extended to generate an outline of the batch submission script, if not its entirety. Further, it can be extended to submit the job, monitor waiting in the queue until execution, then collect the results and display them to the user.

While such an extended Snap! environment may not replace a more traditional HPC workflow, such an approach could serve as a bridge allowing advanced users to transition from learning how to program in parallel with Snap! to using it to support their actual workflow. For many scientists, this may be sufficient enough such that the more sophisticated environments are not necessary.

Another aspect of the Snap! environment that we are working on encompasses the conversion of Snap! programs to textual source code, and in particular, how to map the dynamic types of variables in Snap! to the static types in languages such as C, C++, and Fortran. The type mapping is needed to generate correct source code as well as to achieve good performance.

Finally, we also wish to extend Snap! to extract even more intra-node parallelism as well as support inter-node parallelism.

While our results to date indicate that Snap! is an engaging way to introduce parallel computing, further work needs to be done to rigorously quantify outcomes of using Snap! to teach parallel programming. As mentioned earlier, we are planning to deploy and assess Snap! in a "CS0 Computer Literacy" course [11], consisting primarily of first-year computer science students, in the spring semester of 2017.

7. 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. These 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 [15]. 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 [16].

In [17], 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.

8. Conclusion

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 using `Parallel.js`. 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 scenario demonstrated in this paper, the Snap! blocks are translated into OpenMP, compiled, and executed.

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.

9. Acknowledgement

The work was support in part by NSF ACI-1353786. The authors would also like to acknowledge Eli Tilevich for his 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).
URL `https://www.collegeboard.org/college-board-officially-launches-new-ap-computer-science-principles-course`

[2] The College Board, College Board and NSF Expand Partnership to Bring Computer Science Classes to High Schools Across the U.S. (June 2015).
URL `https://www.nsf.gov/news/news_summ.jsp?cntn_id=135335`

[3] E. Joseph, A. Snell, C. Willard, (July 2004).
URL `http://www.compete.org/storage/images/uploads/File/PDF%20Files/HPC_Users_Survey%202004.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 52 (11) (2009) 60–67.

[5] B. Harvey, D. Garcia, J. Paley, 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, J. Webber, Taxonomy for visual parallel programming languages, Tech. rep., University of Newcastle upon Tyne, Computing Science (2003). URL http://www.cs.ncl.ac.uk/publications/trs/papers/793.pdf

[7] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Communications of the ACM - 50th Anniversary Issue: 1958 - 2008 51 (2008) 107–113.

[8] Apache Software Foundation, Hadoop.
URL http://hadoop.apache.org

[9] W3C, Web workers, w3c working draft 24 september 2015.
URL https://www.w3.org/TR/workers/

[10] A. Savitzky, S. Mayr, Parallel.js.
URL http://adambom.github.com/parallel.js

[11] S. K. Prasad, A. Chtchelkanova, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A. L. Salle, R. LeBlanc, A. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, J. Wu, Nsf/ieee-tcpp curriculum initiative on parallel and distributed computing - core topics for undergraduates, version i, http://www.cs.gsu.edu/~tcpp/curriculum/ (Checked 2016-12-13) (Dec 2012).

[12] L. Meadows, T. Mattson, A hands-on introduction to openmp (November 2008).

[13] A. Silberschatz, P. Galvin, G. Gagne, Operating System Concepts, 9th Edition, Wiley, Hoboken, NJ.

[14] B. Harvey, J. Monig, Snap! reference manual.
URL http://snap.berkeley.edu/SnapManual.pdf

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

[16] National Instruments, LabVIEW System Design Software.
URL http://www.ni.com/labview/

[17] V. Averbukh, M. Bakhterev, The analysis of visual parallel programming languages, Advances in Computer Science: an International Journal 2 (4) (2013) 126–31.