PPP: Parallel Programming with Pictures

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

I. Introduction

Over the past decade, we have seen the increasing ubiquity of multicore computing across every sector of our society. Multicore systems exist not only in supercomputers, but also in desktops, laptops, and even smartphones. Unfortunately, college students are typically not exposed to parallel programming concepts until their senior year, and then, only if they decide to take the elective course. Many do *not* because it is considered "too hard." By this point, they have been "brainwashed" with an ingrained sequential mindset in their approach to writing computer codes. A future crisis is looming as the demand for high-performance computing grows [1], while the development of competent parallel programmers does not keep pace.

The lack of easy-to-use constructs and easily accessible programming environments for parallel computing is perhaps the biggest reason for the belief that explicit parallel programming is "too hard" to undertake. For example, think about the difficulty for a beginning parallel programmer to write, compile, and debug a code in pthreads or OpenMP when writing a program as simple as the "Game of Life." If easier-to-use parallel constructs were available, these constructs would be viewed as key language abstractions, much in the way that `if-then-else` statements and `for` loops are, and they would all be taught alongside each other. By teaching sequential and parallel programming logic at the same time, we avoid creating programmers who can only think sequentially.

By providing appropriate parallel abstractions and programming constructs, students would have the tools necessary for learning the skills and logic necessary for programming in parallel and would be able to do so much sooner than their senior year in college. Block-based languages such as Scratch [2] and Snap! [3] have been successful in teaching novice users (ranging from K-12 to college students to even professional re-trainees) the fundamentals of computer programming by offering unique, easy-to-use, picture-based programming environments that have broad appeal and a low barrier to entry. We leverage the approach taken with these visual languages, so that we can facilitate the necessary forward shift in the time line for introducing parallel constructs, and subsequently, the fundamentals of high-performance computing (HPC).

II. Approach

Here we describe our introduction of parallel constructs into the block-based language Snap!, a visual language that is based on Scratch. Snap!, written in JavaScript, is single-threaded, and it only achieves concurrency by using an event-based, time-sharing, execution model. However, the recently-released HTML5 specification introduces the notable new feature of *Web Workers*. Web Workers enable the spawning of separate compute-intensive JavaScript threads in the background, thereby allowing the browser to remain responsive to the user. Our approach designs an appropriate abstraction of the HTML5 Web Workers within the Snap! interface and incorporates it seamlessly into the existing programming environment. Snap! users would then have an easy-to-use mechanism to enable *truly parallel* programming.

To extend Snap! to use Web Workers, we leverage a toolkit called `parallel.js`. It offers an API that readily allows for the spawning of workers and retrieval of results. Fig. 1 shows a Javascript function for calculating Fibonacci numbers, which is associated with the corresponding Snap! block called `pfib` that accepts an array of numbers as its input, as shown in Fig. 2.

Fig. 3 shows a typical Snap! program that *serially* computes the Fibonacci number for each element of an array in 21 seconds. Fig. 4 shows a similar Snap! program, but this time, computing the Fibonacci numbers *in parallel* by utilizing Web Workers to realize a data-parallel approach (SIMD). The parallel Snap! program completes in about *one* second.

We have also realized other parallel constructs, such as the classical *producer-consumer*. As with the *pfib* block, the

```
Process.prototype.reportParallelFib = function(list) {
    var p;

    function fib(n) {if (n<2) {return 1} else {return fib(n-2) + fib(n-1)}};

    if (this.context.inputs.length < 2) {
        p = new Parallel(list.asArray());
        p.map(fib);
        this.context.inputs[1] = p; // store object in this.context.inputs...
    } else {
        p = this.context.inputs[1]; // ...to check in later runSteps for completion
        if (p.operation._resolved) {
            return new List(p.data);
        };
    }
    this.pushContext('doYield');
    this.pushContext();
};
```

Fig. 1. Internal Snap! function calculates Fibonacci numbers in parallel.

Fig. 2. New block that calculates Fibonacci numbers over an array in parallel.

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

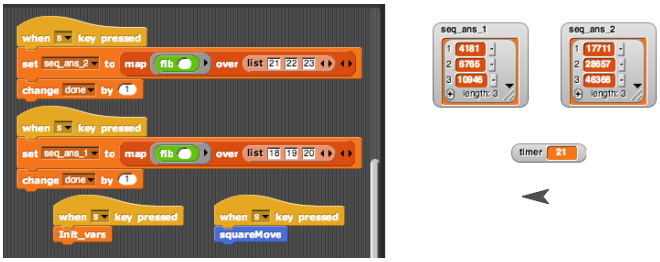

Fig. 3. Serial version of computing Fibonacci numbers.

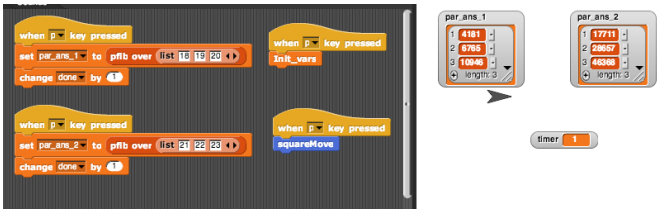

Fig. 4. Parallel version of computing Fibonacci numbers.

producer-consumer construct is supported by the necessary underlying code that abstracts out the low-level details of parallelism. Looking "under the hood," Fig. 5 and Fig. 6 show the Snap! building blocks needed for the producer object and consumer object to interact via a shared buffer, respectively. Fig. 7 shows a realization of the producer-consumer problem with bees as producers and bears as consumers. In the middle is the shared buffer through which the two sets communicate.

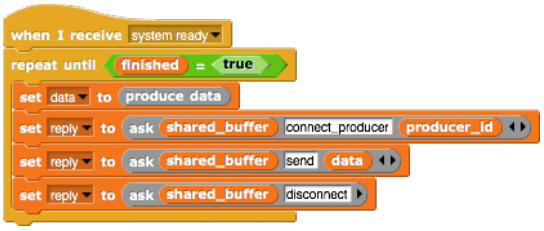

Fig. 5. Producer Object

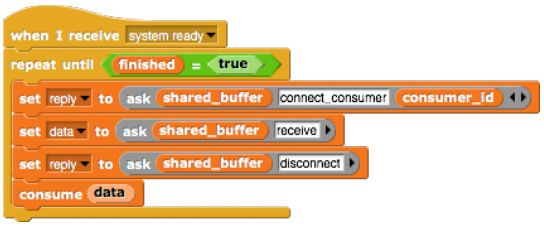

Fig. 6. Consumer Object

The shared buffer and corresponding communication blocks are examples of the types of abstractions for block-based languages that are needed to better enable parallel programming with pictures. The initial back-end for our "Parallel Programming with Pictures" runs on laptops, desktops, and servers, as visually shown in Fig. 7. Future work includes fleshing out the environment to automatically generate parallel code that can run on parallel clusters such as Tiny Titan, and in the longer term, Titan, both at Oak Ridge National Laboratory.

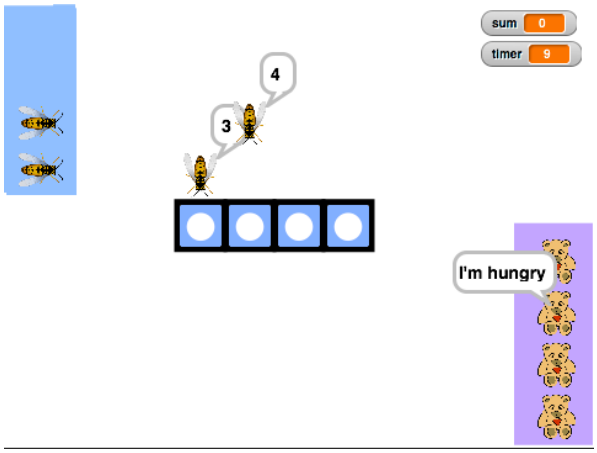

Fig. 7. Consumers depositing data into the shared buffer

III. Conclusions and Future Work

Our work seeks to provide enhancements to Snap! that would allow novice programmers to learn about and use *explicit* parallel constructs much earlier than they otherwise would or could. Coordinating the complex interactions of parallel systems is non-trivial, and our position is that current parallel constructs are not at a high enough level of abstraction to be accessible to the novice user. We view this as an artificial barrier that should be eliminated through the introduction of appropriate abstractions that hide the low-level details that implement an object (such as the shared buffer), along with the methods used to interact with it.

IV. Acknowledgement

The work was support in part by NSF ACI-1353786.

References

[1] B. Obama. (2015) Executive Order – Creating a National Strategic Computing Initiative. [Online]. Available: https://www.whitehouse.gov/the-press-office/2015/07/29/executive-order-creating-national-strategic-computing-initiative

[2] 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.

[3] "Snap!:(Build Your Own Blocks), author=Harvey, Brian and Garcia, Daniel and Paley, Josh and Segars, Luke, booktitle=Proceedings of the 43rd ACM technical symposium on Computer Science Education, pages=662–662, year=2012, organization=ACM."