Block-Based Programming Abstractions for Explicit Parallel Computing

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

***Abstract*—With the majority of computing devices now featuring multiple computing cores, modern programmers need to be able to write programs that utilize these cores in parallel to extract the requisite levels of performance. Despite the need for such *explicit parallel computing*, few programmers are properly groomed in the mindset and the practices of parallel computing. Block-based programming languages, such as Scratch and Snap!, have proven to be a highly effective means of teaching fundamental programming concepts to a wide student audience. Nevertheless, the rich feature-set of mainstream block-based programming environments lack abstractions for explicit parallel programming, thus missing the opportunity to introduce this increasingly important programming concept at a time when the students' minds are most receptive. This paper reports on the results of an NSF-sponsored project for adding and integrating explicit programming abstractions, including producer-consumer, master-worker, and MapReduce, to block-based languages. We describe our reference implementation of adding the producer-consumer abstraction to Snap! and an educational project that utilizes this abstraction. This project clearly demonstrate the key features of parallel processing, without unduly burdening the programmer with the low-level details that this programming model typically entails. Our initial results show great potential in introducing the key concepts of parallel computing via block-based programming.**

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

I. Introduction

The traditional pillars of K-12 education are the so-called three *R's*: *R*eading, w*R*iting, and a*R*ithmetic. We postulate that the three R's above need to be supplemented with a fourth *R*: *R*easoning, or if you will, a renaissance in complex reasoning that embodies computational thinking. 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, particularly parallel computing. Because computing has emerged as a third pillar of science, complementing the traditional pillars of theory and experimentation, it can accelerate discovery and innovation and create a fundamental change in how research, development, and technology transfer in the sciences, engineering, business, humanities, and arts will be conducted in the 21st century. For example, in a study conducted by the U.S. Council of Competitiveness, 97% of surveyed U.S. businesses noted that they could *not* exist or compete without the innovative use of high-performance parallel computing (HPC) [3]. Unfortunately, those same companies lament the dearth of a trained workforce that is familiar with parallel computing concepts.

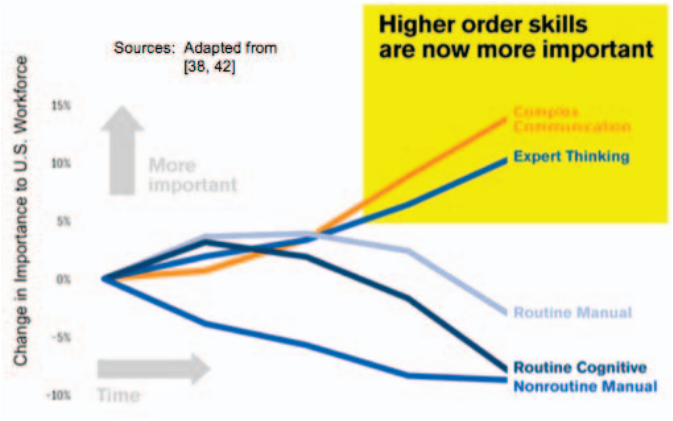

Fig. 1. Technological Changes Affecting U.S. Workforce Skills.

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

To address the need of improving the teaching of parallel computing concepts, we have been pursuing a project whose goal is 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,

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

it may be too late to groom students to think truly in parallel. Instead, we posit that explicit parallel abstractions, such as producer-consumer, should be viewed as fundamental to programming as the `for` loop. By exposing explicit parallel programming via key language abstractions, we aim to harmoniously introduce students to parallel computing from the very start.

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

II. Background

Figure 2 shows the typical Snap! environment. Users program the behavior of actors called *sprites*, which appear on the *stage* area in the upper right. The *palette* area along the left side contains template blocks that users drag and drop into the *scripts* area in the middle. Users connect the blocks together linearly to form programs, which they do for each sprite in the application. When activated, the scripts run and the resulting output of behaviors of the sprites can be observed on the stage. In the screen shot shown in the figure, the sprites are the bees, the bears, the hives, and the honey jars. The displayed script defines the behavior of the bees, which is to perform a little "bee dance," make honey, and deliver it to the bears.

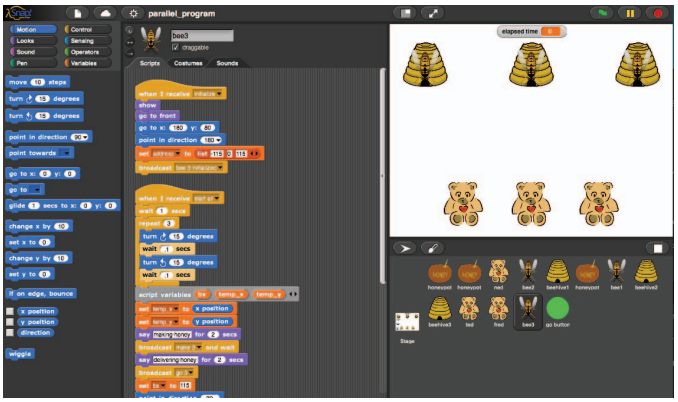

Fig. 2. The Snap! Graphical User Interface.

Each sprite has its own associated set of scripts which run simultaneously when the user clicks on the go button. In this manner, the system supports *implicit* parallelism, in that all sprites execute their scripts concurrently, with the control of the entire application being managed by the underlying JavaScript implementation. However, to code *explicit* parallel behavior that is purposefully coordinated using the Snap! code blocks themselves, while supported in Snap! in a cumbersome and rudimentary fashion, requires a knowledge of computer science concepts that are difficult for novice users to understand and apply successfully in order to correctly achieve the desired behavior.

Many different concurrency paradigms exist, with each pattern tailored to solving a specific kind of problem. For instance, the producer-consumer paradigm is used when the application produces multiple sets of data that must be processed in order, as with a program that handles network communication. The producer task receives incoming data packets from the network interface and stores them to a buffer where consumer tasks retrieve them for further processing. Because processing the data likely takes longer than acquiring it, placing the producer and consumer tasks in separate threads makes such an application much more efficient. Communication between the producer and consumer threads occurs via a shared, finite buffer implemented as a first-in, first-out (FIFO) queue. This creates a loosely-coupled system wherein the producers and the consumers do not need to know about each other beyond the producer knowing that something is taking the data out of the buffer to make room for more and the consumer knowing that something is putting the data in.

Indeed, the producer-consumer problem, also known as the bounded-buffer problem, is defined as one or more producer threads creating data and placing it in a shared and finite buffer, and one or more consumer threads removing the available data and operating on it. Constraints on the system stipulate that a producer with ready data must wait until an empty buffer slot is available before depositing the data, and a consumer ready for new data must wait until it is deposited in the buffer before retrieving it. Access to the shared buffer must be carefully orchestrated, i.e., synchronized, to prevent corruption of the data when multiple threads attempt to update it at the same time.

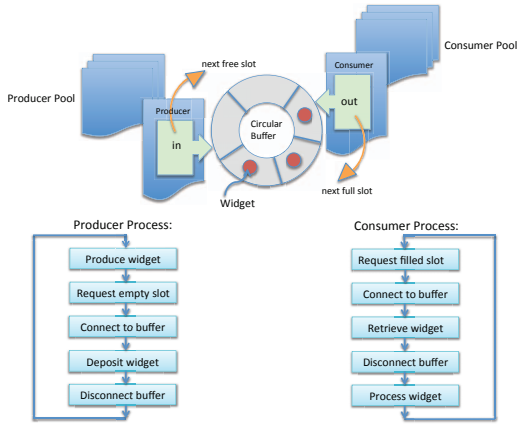

Fig. 3. Producer-Consumer Paradigm.

Figure 3 illustrates the producer-consumer paradigm. It shows the shared FIFO queue as a circular buffer with the head, or *in*, pointer maintaining where the producer is to insert the next data item, and the tail, or *out*, pointer maintaining where the consumer is to retrieve the next available data item to be processed.

Due to their complexity, explicitly parallel programs (in which the user himself defines the coordinating logic) are prone to errors and can be notoriously difficult to debug. Coding such parallel behavior requires a certain sophistication of logic and programming skills that more novice programmers generally do not possess, even though parallel behavior is easy enough to understand intuitively at a high level. We posit that teaching concurrency to computer science students is artificially delayed due to the cumbersome, low-level constructs that are currently available in commonly-used, text-based languages such as POSIX threads (Pthreads) in C. We propose that appropriate programming abstractions for a visual language such as Snap! would allow novice users to implement explicit parallel applications at a much earlier stage, before they become too rooted in the sequential way of thinking about programming.

It is the logical thinking required to produce parallel codes that we wish to promote by making the development of such programs more accessible and less error-prone to the novice user. To achieve this, we seek to provide the tools necessary for learning and employing these important concepts in an age-appropriate manner. Part of our approach involves utilizing a feature intrinsic to Snap! that allows the user to define and add his own code blocks to the Snap! programming environment and to define these new blocks using existing blocks. In computer science parlance, this capability is known as *encapsulation*. The basic idea behind encapsulation is to hide complexity, as it

tends to distract from our understanding of the larger system. As any developer can attest, ”Programs must be written for people to read” [6].

In the next section we present some of the high-level constructs that we have developed for explicit parallel programming in Snap!

III. Parallel Snap!

The first exposure to parallel programming for a computer science student would normally occur in a systems class and would involve writing a solution to the producer-consumer problem in C using POSIX threads (Pthreads), an industry standard for the C programming language for creating and manipulating threads. A typical Pthreads implementation, as shown in Figure 4, will follow along the lines of the producer and consumer patterns illustrated in Figure 3, but with additional code required to perform the task coordination necessary to ensure program correctness through a property called *mutual exclusion*. This property guarantees the condition that only one task at a time accesses the shared buffer.

The Pthreads code in Figure 4 implements the producer-consumer pattern appearing in Figure 3. The solution utilizes a shared, circular buffer and three indices: a `head` index that indicates the next slot for a producer to fill, a `tail` index that indicates the next slot containing data for the consumer to remove, and a `num_items` index that indicates when the buffer is either empty or full. Access to the shared buffer must be coordinated using a construct called a *mutex*, short for *mut*ual *ex*clusion, that implements the thread synchronization needed to protect the shared buffer from potential corruption. To write a more complete and general solution requires significantly more code and a greater degree of programming complexity.

Alas, programming similar explicit parallel behavior in Snap!, while feasible, is not a task achievable by novice users and requires more ingenuity on the part of the programmer to achieve than even the Pthreads version. In order to promote an ease of use of explicit parallel constructs that is on par with learning, say, looping mechanisms or conditional statements, we abstract out the low-level details that are not critical to understanding parallel behavior. These abstractions we present as basic building blocks that hide details not pertinent to understanding explicit parallelism in Snap!

Figure 5 through Figure 9 show a series of screen shots of the Snap! implementation of the producer-consumer problem. Figure 5 shows the program launch with the producer bees at home in the upper left corner of the stage, and the consumer bears at home in the lower right corner. In the middle is the shared buffer through which the bees and the bears communicate. The buffer shown is of size four (4). The white circle indicates an empty buffer slot, whereas a black circle indicates a full slot.

The next frame of the program, shown in Figure 6, shows two producers, having already acquired their ”honey data,” enroute to make their deposit to the shared buffer. As the producers near the slots, the slots sense their presence and a connection between a producer and a slot is made and the data is transferred. The buffer then signals the consumer that data is ready and assigns it the next filled slot. Note the values and locations of the data that the producers are depositing.

Figure 7 shows the producers on their way home after making their deposit, and two consumers already enroute to take the ”honey data.” This Figure 7 also shows that a third consumer, i.e., bear, is ready for data, but as no more buffer slots are ready, that consumer must wait.

In Figure 8 we see that the first two consumers have retrieved their data. Compare the value and location of the data items. In Figure 6 we saw that the producer with value 3 placed that item in the first buffer slot. In looking at Figure 8, we can indeed verify that the consumer retrieved data value 3 from the first slot, and in doing so, the buffer sensed the proximity of the consumer, transferred the data value, and changed its slot appearance from black to white to show that it is now empty and available for another item.

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

#define BUF_SIZE 5

int buffer[BUF_SIZE];
int num_items = 0, head = 0, tail = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t produce = PTHREAD_COND_INITIALIZER;
pthread_cond_t consume = PTHREAD_COND_INITIALIZER;

void* producer(void *ptr) {
    int i, data;

    for (i = 1; i <= 2; i++) {
        data = (rand() % 10);                    // Produce
        sleep(data);
        printf("Produced: %d\n", data);
        pthread_mutex_lock(&mutex);              // Request
        if (num_items == BUF_SIZE) {
            pthread_cond_wait(&produce, &mutex); // Connect
        }
        buffer[head] = data;                     // Deposit
        head = (head+1) % BUF_SIZE;
        num_items++;
        pthread_cond_signal(&consume);
        pthread_mutex_unlock(&mutex);            // Disconnect
    }
    pthread_exit(0);
}

void* consumer(void *ptr) {
    int i, data;

    for (i = 1; i <= 2; i++) {
        sleep(rand() % 3);
        pthread_mutex_lock(&mutex);              // Request
        while (num_items == 0)
            pthread_cond_wait(&consume, &mutex); // Connect
        data = buffer[tail];                     // Retrieve
        tail = (tail+1) % BUF_SIZE;
        num_items--;
        pthread_cond_signal(&produce);
        pthread_mutex_unlock(&mutex);            // Disconnect
        printf("Consumed: %d\n", data);          // Consume
    }
    pthread_exit(0);
}

int main(int argc, char **argv) {
    pthread_t p1, p2, p3, p4;
    pthread_t c1, c2, c3, c4;
    int pid = 1, cid = 1;

    pthread_create(&c1, NULL, consumer, NULL);
    pthread_create(&c2, NULL, consumer, NULL);
    pthread_create(&c3, NULL, consumer, NULL);
    pthread_create(&c4, NULL, consumer, NULL);
    pthread_create(&p1, NULL, producer, NULL);
    pthread_create(&p2, NULL, producer, NULL);
    pthread_create(&p3, NULL, producer, NULL);
    pthread_create(&p4, NULL, producer, NULL);
    pthread_exit(0);
}
```

Fig. 4. Solution to Producer-Consumer Problem Using Pthreads.

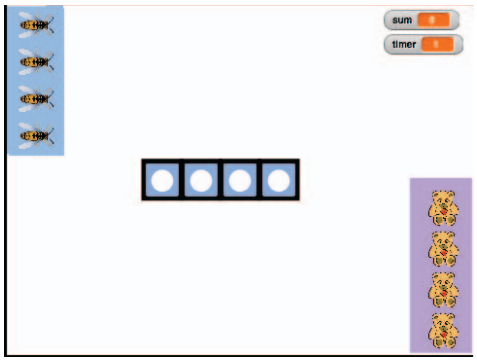

Fig. 5. Launch of the Snap! Producer-Consumer Program.

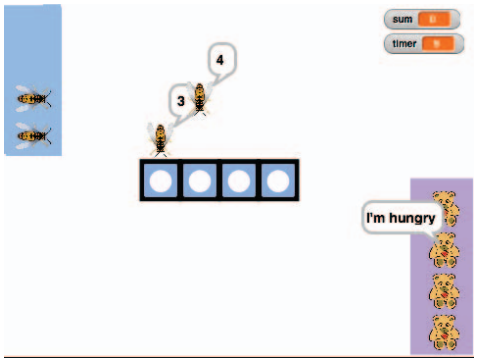

Fig. 6. Bees Making a Deposit.

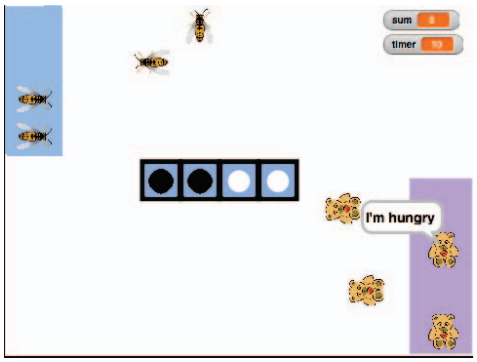

Fig. 7. Bears Retrieving "Honey Data."

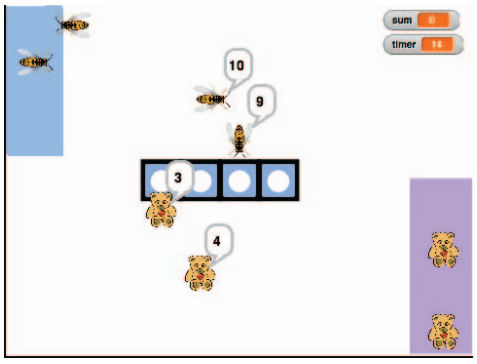

Fig. 8. Bees and Bears Coordinating Access to the Shared Buffer.

Figure 9 displays the final frame of the program showing the last consumers returning home to consume their data.

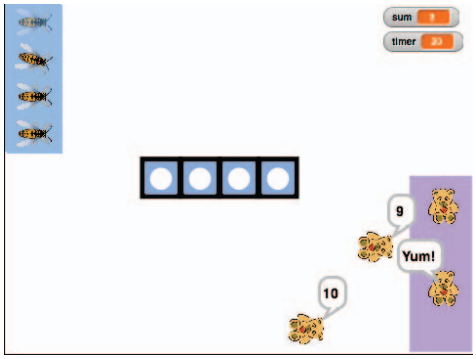

Fig. 9. Bear Consuming Honey.

Below we discuss the details of our approach for introducing explicit parallelism to Snap! When the program starts, the system is initialized, and when the buffer is ready, it signals the bees and bears to begin. The iterative behavior of the bees follows the normal producer pattern: (1) produce data, (2) request empty buffer slot, (3) acquire slot, (4) deposit data, and (5) repeat. The bears are similarly coded to realize the normal consumer pattern: (1) request filled slot, (2) acquire slot, (3) retrieve data, (4) consume data, and (5) repeat. The blocks implementing the basic solution for the bees and the bears are interspersed with code blocks that achieve the various animation effects. The potential for Snap! to visually illustrate parallel systems behavior through animation makes it particularly desirable as a teaching tool for children, given age-appropriate examples such as "the bees and the bears" demo.

Fig. 10. New "make buffer" Definition for Snap!

For the preceding example, the shared buffer has a capacity of four (4) slots; however this can be changed programmatically to accommodate any size buffer between 1 and 10. Figure 10 shows the special reporter block, `make buffer`, that was introduced to encapsulate the implementation details of the buffer. The `make buffer` block returns a buffer object containing five methods that can be called on it, namely

1) start
2) connect_producer
3) send
4) connect_consumer
5) receive

Figure 11 shows the code block that a producer with id `id` would use to send a `connect_producer` request to the buffer.

Fig. 11. Calling the `connect_producer` Method on the Buffer Object.

The shared buffer is a perfect example of the types of abstractions that we seek to develop for block-based languages such as Snap! The programmer need not understand the actual implementation details of the buffer in order to use it; the abstraction is at an appropriate

level and provides a general solution that can be reused in other programming scenarios.

IV. Conclusions and Future Work

Our work seeks to provide enhancements to the Snap! visual programming language that would allow novice programmers to learn about and to define more sophisticated behaviors in their applications using *explicit* parallel constructs at an earlier point in their programming careers. Our position is that current parallel constructs are not at a high enough level of abstraction to be accessible to the novice user and that this is an artificial barrier that should be eliminated.

Coordinating the complex interactions of parallel systems is non-trivial, and our initial work shows promise in our approach. As proof of concept, our implementation of the producer-consumer paradigm in Snap! successfully demonstrates that explicit parallel behavior can indeed be achieved in a block-based language such as Snap! The key to making it accessible to non-expert users is to provide abstractions at an appropriate level, such as the "shared buffer" object, and to make sure that such solutions are general enough to be useful in virtually any given programming scenario. A visual language such as Snap! has broad appeal and the costumes and animations give it the capacity to introduce an element of whimsy into any programming exercise.

However, one must be careful not to overlook the more serious applications that languages such as Snap! can facilitate. As an example, instead of the preceding "bees and bears" producer-consumer demo, what if we sought to implement, say, a package delivery system. Instead of bees and hives, we have human workers and warehouses, respectively, and instead of buffers and bears, we have delivery trucks and businesses, respectively. Now we are looking at the potential for modeling serious real-world, grown-up applications such as a product distribution network for a retail store or a supply-and-demand chain for a global enterprise.

V. Acknowledgement

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

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] G. Sussman, H. Abelson, and J. Sussman, "Structure and interpretation of computer programs," 1983.