EcoSim: A Language and Experience Teaching Parallel Programming in Elementary School

Chris Gregg Luther Tychonievich James Cohoon Kim Hazelwood

Department of Computer Science
University of Virginia
PO Box 400740
Charlottesville, VA 22904

ABSTRACT

Traditional introductory programming classes teach sequential programming using a single-threaded programming model. It is typical to wait until a student has developed proficiency in sequential programming before teaching parallel programming. As computer hardware becomes increasingly parallel, there is a greater need for software engineers who are proficient in designing parallel programs, and not just by "parallelizing" sequential designs. Teaching parallelism first is an important step towards educating tomorrow's programmers.

We present an overview of a five-day introductory parallel programming course. We taught the course to nine and ten year-olds with no prior programming experience. Our course utilized a fundamentally parallel language we designed for the course, one with a near-natural language syntax that exposed the parallel processors throughout the code. This language, coupled with an interactive online programming environment, allowed us to teach a wide range of parallel programming concepts in a very limited timeframe.

We also present examples of student-written code that demonstrates their understanding of some basic parallel programming concepts, and we describe the overall course goal and specific lesson plans geared towards teaching students how to "think parallel."

Categories and Subject Descriptors

D.3.2 [**Concurrent Programming**]: Language Classifications—*concurrent, distributed, and parallel languages*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education, curriculum*

General Terms

Languages, Design, Human Factors

Keywords

Concurrent languages, parallel languages, instructional design, introductory programming, pedagogy, education, readability, elementary school, K-12

SIGCSE'12, February 29–March 3, 2012, Raleigh, North Carolina, USA.

1. INTRODUCTION

Introductory programming classes are often taught using languages designed primarily for single-threaded applications. Multi-threaded or parallel programming concepts are considered advanced. It is rare that students learn about parallel programming before a second or third programming course. Traditionally, many colleges and universities in the U.S. provided only a single parallel programming course, often as a senior-level undergraduate elective. When students do learn parallel programming, many have difficulties transitioning from a sequential-programming mentality to a parallel programming mentality. One manifestation of this difficulty is that parallel programming is considered "hard" by many students and instructors alike [9].

As parallel platforms become more common, parallel programming skills are becoming increasingly important. Within the last five years, multicore computing has become the *de facto* standard on desktops and laptops. General Purpose GPU (GPGPU) computing has matured such that multi-core GPUs can be programmed with minimal extensions to traditional languages such as C++ and Python [5]. These trends are expected to continue in coming years [7]. To take full advantage of this increasing parallelism, programmers must understand traditional parallel concepts such as race conditions, atomicity, synchronization, and deadlock. Beyond those skills, they must also be able to look at computing problems and devise solutions that utilize parallel processes. Parallel programming requires a change in mindset as well as practice, adopting new models of what activities are "easy" and "hard" to guide the development of algorithms that can be executed efficiently by highly-parallel hardware.

Rather than strive to correct existing sequential programming habits, we developed an introductory parallel programming course specifically targeting novice programmers. The class, titled "Programming the Computers of the Future," was offered during a five-day weekend enrichment program to students from the 4th and 5th grade (9 and 10 years old). The choice of this age group was primarily based on the opportunity we had to teach novice programmers, but pedagogically the methodology we describe could be applied to any group of beginner programmers. None of the students in our classes had significant prior programming experience. Each class period was two hours long. A week elapsed between each class, during which the students could access the programming development environment online to continue learning independently, although no out-of-classroom assignments were given.

Our course was designed around three objectives.

```
1: a plant has
2:   a position
3:   size, a number
4:   a color
5:
6: create 10 plant and for each
7:   do in order
8:     replace the plant's color with green
9:     replace the plant's size with 10
```

Figure 1: A simple *EcoSim* program to define and create ten green "plants" on the screen.

- Introduce the students to basic parallel programming ideas using multiple processors.
- Help the students develop a mental model of what programming is and what computers do.
- Teach the students to "think parallel" about computing problems.

We measured these objectives through exit surveys, informal interactions with individual students, and by inspecting the code produced by each student. Structured assessment was not possible within the enrichment program structure.

The course used a programming language of our own design, called *EcoSim*[1]. *EcoSim* uses an English-like syntax and makes parallelism both pervasive and visible to the user. We also developed an online environment in which students could write and run code from any computer. This environment gives each students step-by-step feedback on the behavior of each emulated processor. Figure 1 shows an example *EcoSim* program that defines and draws ten green "plants" on the screen, where the plants are represented by circles of radius 10. Figure 4 shows the *EcoSim* development environment, which includes a code window, settings, a console window with output messages, and a window for graphical objects.

2. ECOSIM

We looked for four characteristics in the language we used in our course[2]:

Visibly Parallel: We wanted a language that was parallel unless requested otherwise. We also wanted the idea of parallel processors executing the instructions to be emphasized throughout the language and visible in the runtime environment.

Self-Explanatory: We did not want to have to translate what code meant. We didn't want to re-define symbols like "=" or to introduce new symbols or words. We also wanted all elements of the language to be accessible at the 4th-grade level.

Web-Based: We wanted the students to be able to work at home without installation. This meant building on top of either Javascript or Flash, the only tools we could count on all the students having already installed.

Engaging: We wanted every program the students wrote to interest them. Every program should be graphical, with text reserved for diagnostic information.

[1] *EcoSim* is so-named because we envisioned students using it to program *eco*logical *sim*ulations.

[2] We considered, but rejected, a fifth characteristic: a "complete" language. As an enrichment course, we did not anticipate being a gateway into serious software development. In hindsight, this may have been a poor choice.

Since we were aware of no language having all four desired characteristics, we created our own. We designed the syntax and semantics, wrote a type-checking parser, interpreter, and runtime environment in Javascript. We also created an interactive environment using HTML and CSS, with the HTML5 Canvas element providing graphical output.

2.1 The EcoSim Runtime

The *EcoSim* environment has three basic interfaces: the code entry pane, a status window listing the results of parsing and the behavior of each processor, and a graphical display of the state of each object. The graphical display automatically draws a circle for each object that had a defined position and size in the student's code. By automating the display we were able to focus on core programming concepts rather than teaching about a graphics API.

In addition to the user interface, the *EcoSim* runtime provides the following:

A fixed number of virtual processors.
At each processing step the interpreter assigns to each processor a task; this task is displayed both in the code pane and in the status window.

A shared work queue for ongoing tasks.
Processors pull jobs off this queue in a random order and insert any unfinished work back on the queue at the end of each step.

A global list of objects of each type.
Each object is placed on a global list of objects of its type when it is instantiated. These lists are used to handle "`for each`" and "`for some`" constructs.

A collision tracker and set of collision handlers.
The code may provide handlers for collisions of objects with position and size. These are given to the processors if the work queue is empty.

A set of idle tasks.
If the work queue is empty and there are no collisions to handle, the remaining processors are given jobs from a set of low-priority tasks.

The runtime is aware of four types: number, color (HTML-compliant color names), comparison (boolean values; used only behind the scenes to type-check guard expressions), and position (a pair of numbers, x and y). User-defined types are built out of these parts.

2.2 The EcoSim Language

Three principles guided our design of *EcoSim*'s language constructs: "audience: processor", "self-describing syntax", and "anonymous by default." The first two are directly parallel our first two objectives: we wanted the processors to be visible throughout the language and the language to be self-explanatory. We added the third principal, defaulting to anonymity, to reduce students' need to come up with names for variables, simplifying the programming process. We hoped it would also reduce the likelihood of conflict over limited, named computing resources, but this kind of conflict did not arise in any class example.

Rather than provide a full description of the language that grew out of these principles, we provide a few elements that typify our design choices and rely on *EcoSim*'s self-describing character to render later code examples understandable.

2.2.1 Statements

The first operator we considered was the assignment operator. We had discovered in previous interactions with CS1 students that the syntax used in "`x = x + 1`" was often confusing. We brainstormed ways we might explain that operation, things like "x is redefined; it's new value is 1 + the old value of x," but most were too verbose or failed the "audience: processor" principle. We finally settled on "`replace x with old x + 1`", which we implemented as two rules: an assignment syntax of "`replace` *lvalue* `with` *rvalue*" and a requirement that the word "`old`" precede rvalue variables that also appear in the lvalue.

Similar processes resulted in "`while(1 < 2)`" becoming "`as long as 1 < 2`", "`else`" being replaced with "`otherwise`", and "`double x = 3`" becoming "`start x as 3`". We replaced the membership operator (commonly "`.`" or "`->`") with the more English-like "`'s`" (as in "`baz's position's x`"). We also implemented type inference to create a statically-typed language without needing to declare variable types.

The runtime keeps a list of objects of each type (see §2.1); we add to these lists by writing "`create` *number type*" and remove with a "`destroy`" command. We can access objects either by grabbing one randomly selected object:

```
for some number
  destroy the number
```

or by accessing all of them in parallel:

```
for each number
  replace the number with the old number + 1
```

2.2.2 Blocks

Blocks caused us some difficulty with our "self-explanatory" goal because they largely don't exist in spoken language. We decided to go with indentation as more readable than delimited blocks, but neither we nor our students were entirely happy with this choice.

The basic notion behind a block is "do each of these things once", followed by a list of statements. We distinguished different kinds of blocks based on their sequentiality and atomicity. This gives the four blocks types "`do in order`", "`do in any order`", "`do atomically in order`", and "`do atomically in any order`". Atomic blocks implicitly lock all of the variables they accesses. "In order" blocks can handle only a single processor, while multiple processors can execute the statements of an "in any order" block simultaneously.

Examples of blocks are provided in Figures 1, 2, and 3.

2.2.3 Definitions

We observed that structures, properties, and subroutine definitions are describing "what we mean by X" rather than "you should do X". We thus decided that, per the "audience: processor" principle, we should word these to inform, not direct, the processor.

For example, to introduce a structure type named "plant" with a named field "size", an anonymous position field, and two anonymous colors, we write

```
a plant has
  size, a number
  a position
  2 color
```

We can also define properties for plants:

```
a plant's age is the plant's size − 5
a plant's trunk is the plant's 2nd color
```

Properties are always single expressions and are accessed exactly like fields. Multiple fields (such as "`color`" above) can only be accessed by compile-time ordinals like "`5th`"; you cannot write "`the plant's` *n*`th color`" for variable n.

Subroutines are defined with a "how to":

```
how to add a number years to a plant
  replace the plant's size with the plant's old size
      + the number
```

Calling a subroutine is straightforward:

```
for some plant
  add 3 years to the plant
```

Because this syntax for defining and calling subroutines is context sensitive, it is not easy to achieve using most parser techniques. Our parser type-checks and builds the symbol table as it goes, so when we parse a line "`how to` *words*" we already know which of the words identify types and which are unbound words naming the subroutine. Similarly, we know from context that "`the plant`" is a value and thus that we are calling a method named "add ___ years to ___" and not "add ___ years to the plant".

2.2.4 Handlers

The last element of the language we want to identify is collision handlers and idle operations.

```
when a bulldozer hits a plant
  destroy the plant
when bored
  create a plant
```

Again, these are worded to address the processor in a self-explaining way. They prevent the need for an explicit simulation loop and make event-oriented programming straightforward. Multiple "`when bored`" declarations were permitted, with the runtime selecting between them at random.

The general structure of our collision handlers would have worked with arbitrary predicates, such as "`when a plant's size > 7`". We chose not to expose this generality to the students because they could easily write their own single-object predicates already and collisions covered all of the multi-object interactions we expected in an ecological simulation.

3. COURSE OVERVIEW

The pilot course we created was for fourth and fifth grade students in an enrichment program that is run through our university. We designed the course and *EcoSim* concurrently, for an audience of self-selected primary school students with no prior formal programming experience.

3.1 Ecosystem in Parallel

The original conception of the pilot course was, simply, "Let's teach fourth and fifth graders about parallel programming." To streamline our lessons we decided to structure our examples around a single theme. The theme we selected was "ecosystems" because it is familiar to students at that level, inherently parallel, easy to visualize, and allows for a wide range of example applications. By the end of the course students had simulated ecosystems ranging from simple growing plants to complex interactions between diurnal locally-seeding plants, competing herbivores and hunters, carnivorous and poisonous plants, and other more imaginative fantasy elements.

Students quickly learned the importance of initial conditions and parameters, both from a computational perspective and a scientific one. For example, students found that starting ten thousand herbivores in a field with only ten plants

both results in over-grazing and starvation and slows the computer to a crawl as it looks for each object's collisions. We spent a number of classes discussing the remote St. Matthew Island in Alaska, where a herd of reindeer overpopulated and subsequently died out [14, 17]. With the *EcoSim* model the students were able to adjust the parameters and look for an equilibrium that would have allowed the reindeer to survive.

3.2 Getting the Students to "Think Parallel"

We began each class period with a conversation and activity introducing a new parallel programming concept. Table 1 shows the group activities we conducted and their associated parallel processing concept or concepts. During and after each activity, we discussed the associated concept, and in most cases we then wrote a simple program in *EcoSim* that demonstrated the idea.

For example, on the first day of class we introduced the students to the difference in computational time between parallel and sequential processes by having them sort themselves by height. First, we allowed the students to line themselves up by height, all at once (the parallel method), and we timed this; it took roughly forty-five seconds for a class of eighteen. Next, we re-randomized the class and assigned one student to be the "processor," in charge of sorting the students. Unsurprisingly, this took roughly twice as long, leading to a fruitful discussion on why parallel processing can be faster. We then showed the students that *EcoSim* allows a programmer to set the number of processors that will be used to run the program and had the students experiment with the speedups resulting from multiple processors.

After the opening class discussion we would transition to programming exercises. Each student would sit at a computer with *EcoSim* loaded into their web browser, and they were able to type out the examples as we wrote them on the overhead projector. For example, after the race condition activity, we wrote the programs in Figure 2. These programs demonstrate a race condition stemming from allowing multiple processors to replace the moths' color in an arbitrary order. When the "In order" program runs, all of the moths end up black at the end, but when the "In any order" program runs, an indeterminate number of moths end up black and the remainder are gray. This outcome was unexpected for many of the students, and we allowed them to experiment with the code, trying out other conditions to explore the idea more fully.

The ability to step through code in the *EcoSim* environment allowed students to see how the computer followed their instructions. For example, if *EcoSim* is set to use two processors for the "In order" program from Figure 2, the students see that individual "moths" changing color two at a time. Often when students were surprised at some result of program behavior we would step though the code, discussing how each processor's actions effected the simulation.

3.3 The Use of Example Programs

As with any programming course, example programs played an important role in teaching our course. Because this course was the first time most of the students had seen any programming language at all, we decided to provide a scaffolding in the form of example programs that they could look at and modify. *EcoSim* has a "Load Example" button that brings up a listing of programs written by the instructors. Many times during class we would have students pay attention to the projector as we typed in the code for a program. Once they saw our initial work, we would have them load the example

In order:

```
1: a moth has
2:   a position
3:   a color
4: a moth's size is 50
5:
6: create 10 moth and for each
7:   do in order
8:     replace the moth's color with gray
9:     replace the moth's color with black
```

In any order:

```
1: a moth has
2:   a position
3:   a color
4: a moth's size is 50
5:
6: create 10 moth and for each
7:   do in any order
8:     replace the moth's color with gray
9:     replace the moth's color with black
```

Figure 2: Example *EcoSim* programs that demonstrate race conditions. In the "in order" program, all moths end up black, while in the "out of order" program the final color is dependent on a race condition.

instead of typing it out. This provision saved time (not all pre-teens are fast typists), and it also allowed us to start the whole class at the same point in a program's development. In some cases we gave them example programs that were missing a line or two and asked them to fill in the details themselves. We encouraged the students to modify the programs as well. Students that completed programming tasks before others in the class were able to modify the example programs, write their own programs, or return to programs they had been working on previously.

Another reason we relied on example programs was to build a compendium of programs that the students could go back and look at if they did not remember the details of a particular topic. Frequently, we would direct the students to previously covered examples.

3.4 Student Assessment

Because this course was an ungraded enrichment class, we did not perform formal assessments (e.g., tests or deadline-based homework). However, we reviewed the students' work regularly and gave feedback often. At the beginning of each hour of class, we asked whether anyone had something they wanted to show the class. Many students enjoyed displaying their work on the projector and seemed motivated to create interesting programs they could share with the rest of the class. Some students showed off work that they completed on their own at home during the week between classes, which we highly encouraged.

All student work is captured in a MySQL database that we were able to review regularly. Each time a student clicks on the "Setup Code" (which parses the code and reports errors), the current program is saved, and all versions are retained. Therefore, we were able to look at a students progress, including how many attempts they made at fixing syntax errors and how they went about writing their programs. We used this analysis to determine where we needed to review; e.g., once we realized how much trouble the students had with understanding indentation and blocks, we modified our lesson plan to include a review and further examples.

Concept	Group Activity
Parallel speedup	Students sort themselves, and then one student sorts everyone.
Locks / Atomicity	Everyone shares a pen to write on the whiteboard to increment a number.
Race Conditions	Students roll dice until they get a six, read a number off the board, increment it, roll more dice, and call out the new number, which is written on the board.
Divide and Conquer	All students start with a number, and half hand to their neighbor to add together. This continues until one student has the total sum.

Table 1: Group activities.

```
23: when bored
24:   for each pacman
25:     do in any order
26:       move the pacman's position a random number
               between 10 and 50 right
27:       move the pacman's position a random number
               between 10 and 50 left
28:       move the pacman's position a random number
               between 10 and 50 up
29:       move the pacman's position a random number
               between 10 and 50 down
30:
31: when a pacman hits a ghost
32:   destroy the pacman
```

Figure 3: Part of one student's "Pacman" program, that demonstrates a competent understanding of the programming language and its parallel programming structure.

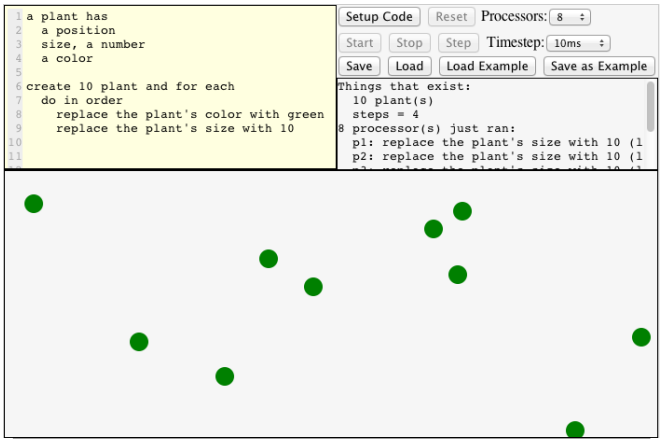

Figure 4: The *EcoSim* web-based integrated development environment hosted at `http://ecosimulation.com`. Code is written and debugged in the top left window, settings are on the top right, a console with runtime and debug information is below the settings, and the main window shows the graphical output of the program.

4. STUDENT WORK AND OUTCOMES

During the course the students had an opportunity to modify example programs and write their own applications. We analyzed the database of student programs to perform basic analysis of each student's programming process. Students worked on between twelve and thirty-two separate programs each, with a mean of twenty programs per student. We observed that the students who worked on more programs tended to be the ones who picked up the material the fastest in class, and were also more likely to work on programs at home. The students' programs ranged from simple five-line examples to over 150-line simulations of complex ecosystems. In addition, some students created novel programs that demonstrated proficient understanding of the language. Figure 3 shows part of one student's novel "Pacman" program, demonstrating that some students were able to use the programming elements discussed in class in a novel, parallel context.

As with any introductory programming course, students can get frustrated with the debugging process. When a student clicks on the "Setup Code" button in *EcoSim*, the parser reports the first parse error (if one or more exist) and highlights the offending line number. Over the duration of the course, it took the students an average of 4.2 attempts to fix a program that had errors. If a student gave up on a program with errors by starting or opening a different program, he or she attempted an average of 8.3 times to fix the error before giving up.

On the last day of class, the students demonstrated their work to their parents and siblings. We had each student to prepare a program that showcased their own work—either one they created on their own or an example from class that they modified significantly. They enthusiastically demonstrated their programs, including various modifications to the St. Matthew Island ecosystem, programs that had snow "fall" on the screen and accumulate at the bottom, simulations of mice searching for cheese, etc.

5. RELATED WORK

Parallel computing is not new, dating back to 1955 and the IBM 704 [12]. Today, multicore desktop and laptop computers are ubiquitous—when novice programmers write their first code today, it is often running on a parallel computer. In order to make the most efficient use of these computers, parallel programming is necessary.

There are numerous programming languages available for desktop parallel programming. Many of these languages are extensions, libraries, or APIs built on top of sequential languages such as C or Fortran (e.g., OpenMP, CUDA, OpenCL, Intel Thread Building Blocks, pthreads, Cilk, Co-array Fortran, and Unified Parallel C), requiring a novice programmer to first become proficient in a sequential language before tackling the parallel programming concepts. While this does not necessarily hinder a student's overall programming ability, parallel programming tends to receive less emphasis than simply learning the sequential aspects of the language.

There are languages designed specifically for parallel programming, but they tend have advanced syntax and are targeted towards students already proficient at programming in general (e.g., X10 [8], Go [4], and ParaSail [21]). Novice programmers would almost certainly find it difficult to learn one of these languages as an introduction to programming.

Several researchers have investigated teaching parallel programming concepts at the undergraduate level [13, 18, 23] and at least one at the secondary school level [22]. These stud-

ies targeted both first and second year students using openly available parallel programming languages (e.g., OpenMP, MPI, and CUDA). A common result was that students were motivated more by concrete examples of parallel code and significant classroom programming time rather than by abstract concepts delivered in a lecture. We likewise utilized concrete examples and in-class programming time.

Parallel concepts could be taught using groups of robots. However, robotics packages aimed at younger students such as Mindstorms [19], PicoCricket [2], and Arduino [1] focus on single-processor single-robot programming tasks.

Several successful languages have been designed to have English-like syntax, including COBOL [3], AppleScript [6], and Wolfram Alpha [15]. Like *EcoSim*, each was designed to accessible to non-programmers and each targets a relatively narrow set of possible use cases. None are targeted toward teaching parallel programming concepts.

An alternative to "readable" languages is to use a graphical rather than textual representation of the program. Recent examples of graphically-represented languages that are aimed at children include Alice [10], Scratch [16], Greenfoot [11], and Kodu [20]. While an exhaustive survey of graphical programming languages was impractical, we were unaware of any that did not require installation and were suitable for teaching core parallel programming concepts.

6. CONCLUSIONS

We have demonstrated that it is possible to teach elementary school students parallel programming, even if they are completely novice programmers. We were able to teach basic programming skills as well as concepts such as race conditions, atomicity, locks, and the speedups that can be obtained by using multiple processors. None of our students had any difficulty with parallelism, and most were able to explain core parallel concepts to their parents at the end of the course.

To facilitate learning in our course, we

- designed a unique, English-like parallel programming language and associated visual web interface;
- provided a curriculum that taught programming through a parallel lens; and
- utilized learning activities, think-pair-share discussions, and individual guided exploration to teach each concept.

We found the students eagerly learned the *EcoSim* language. Despite some trouble with the notion of indented blocks, most students were able to successfully write significant programs without difficulty.

Parallel programming does not have to be difficult to teach or learn, and it is a skill that will become even more important as computers continue to become more parallel in the future.

Acknowledgments

This work was supported in part by NSF grants BPC-0739259, BPC-104282, CNS-0747273, CSR-0916908, and CNS-0964627; and awards from Microsoft and Google. We would like to thank the University of Virginia Saturday and Summer Enrichment Program for the opportunity to teach *Programming the Computers of the Future* during the winter 2011 session. We are grateful to the students in the class for their enthusiasm for the subject. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

7. REFERENCES

[1] Arduino. Available: `http://arduino.cc`.

[2] Picocricket. Available: `http://picocricket.com`.

[3] *General Information Manual, IBM Commercial Translator*. IBM Corporation, New York, 1959.

[4] The go programming language. `http://golang.org`, accessed on August 6, 2011.

[5] Archives for programming languages. `http://gpgpu.org/tag/programming-languages`, accessed on July 31, 2011.

[6] Apple, Inc. Applescript overview. `http://developer.apple.com/library/mac/documentation/AppleScript/Conceptual/AppleScriptX/AppleScriptX.pdf`, accessed on August 9, 2011.

[7] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communcations of the ACM*, 52:56–67, October 2009.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[9] C. R. Cook, C. M. Pancake, and R. Walpole. Are expectations for parallelism too high?: a survey of potential parallel users. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 126–133. ACM, 1994.

[10] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-d tool for introductory programming concepts. *J. Comput. Small Coll.*, 15:107–116, April 2000.

[11] P. Henrickson. A direct interaction tool for object-oriented programming education. Master's thesis, University of Southen Denmark, Maersk Mc-Kinney Moller Inst. for Production Technology, 2004.

[12] R. Hockney and C. Jesshope. *Parallel computers 2: Architecture, programming, and algorithms*, volume 2. Taylor & Francis, 1988.

[13] D. Johnson, D. Kotz, and F. Makedon. Teaching parallel computing to freshman. In *Conference on Parallel Computing for Undergraduates*, pages 1–7, 1994.

[14] D. Klein. The introduction, increase, and crash of reindeer on St. Matthew Island. *The Journal of Wildlife Management*, pages 350–367, 1968.

[15] W. A. LLC. Wolfram|Alpha: Computational knowledge engine. `http://www.wolframalpha.com`, accessed on August 9, 2011.

[16] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education*, November 2010.

[17] S. McMillen. St. matthew island. `http://www.recombinantrecords.net/2011/02/09/st-matthew-island`, accessed on August 6, 2011.

[18] C. Nevison. Parallel computing in the undergraduate curriculum. *Computer*, 28(12):51–56, December 1995.

[19] S. Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, 1980.

[20] K. T. Steele. Kodu language and grammar specification. `http://research.microsoft.com/en-us/projects/kodu/kodugrammar.pdf`, August 2010.

[21] S. T. Taft. Designing parasail, a new programming language. `http://parasail-programming-language.blogspot.com`, accessed on Auguet 9, 2011.

[22] S. Torbert, U. Vishkin, R. Tzur, and D. J. Ellison. Is teaching parallel algorithmic thinking to high school students possible?: one teacher's experience. In *Proceedings of the 41st ACM technical symposium on computer science education*, SIGCSE '10, pages 290–294. ACM, 2010.

[23] J. Tourino, M. Martin, J. Tarrio, and M. Arenaz. A grid portal for an undergraduate parallel programming course. *Education, IEEE Transactions on*, 48(3):391–399, aug. 2005.