Parallel Programming in Elementary School

Chris Gregg
University of Virginia
Department of Computer Science
Charlottesville, VA

Luther Tychonievich
University of Virginia
Department of Computer Science
Charlottesville, VA

Kim Hazelwood
University of Virginia
Department of Computer Science
Charlottesville, VA

James Cohoon
University of Virginia
Department of Computer Science
Charlottesville, VA

ABSTRACT

Traditional introductory programming classes focus on teaching sequential programming skills using conventional programming languages and single-threaded applications. It isn't generally until much later in a student's programming education that he or she learns about parallel programming and associated topics such as race conditions, locks, or data consistency. With the increased popularity of multicore CPUs and GPUs capable of GPGPU computing, there is a greater need for programmers who are not only proficient in parallel programming, but who are not burdened by an inclination towards trying to solve a problem in a sequential fashion, with parallelism tacked on as an afterthought.

Pedagogically, there is a case to be made that teaching parallelism first is an important step towards educating tomorrow's programmers for the challenges of programming multicore and GPGPU systems. We present an overview of a five-day introductory parallel programming course we taught to a group of nine and ten year-olds, using a near-natural language syntax parallel programming language we created, targeted towards students with no previous programming experience. Our language is simple but powerful and consists of a simulated parallel programming environment and the ability to run or step through programs.

We provide 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

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Languages, Design

Keywords

Concurrent, distributed, and parallel languages, Instructional Design, Introductory Programming, Pedagogy, Education

1. INTRODUCTION

Introductory programming classes are almost universally taught using languages designed primarily for single-threaded applications. Multi-threaded or parallel programming concepts are considered advanced, and it is rare that students learn about parallel programming before a second or third programming course. Indeed, most colleges and universities in the United States provide a single parallel programming course available to upper level undergraduates or graduate students{FIXME citestats}, and such courses are almost always optional in the computer science curriculum. In many cases, only students who are interested in high performance computing are ever exposed to parallel programming, and the average programmer never receives any traditional instruction in parallel programming at all. Additionally, when students do learn parallel programming, many have difficulties transitioning from a sequential-programming mentality to a parallel programming mentality, especially as parallel programming is considered "hard" by many students and instructors alike. [7]

Within the last five years, multicore computing has become the *de facto* standard on desktops and laptops, and 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 [2]. The trend towards increasing cores to program on a single machine does not show any signs of abating in the near future [4], and therefore parallel programming skills are going to become increasingly important. Programmers must not only thoroughly understand parallel programming concepts such as race conditions, atomicity, synchronization,

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

create 10 plant and for each
    do in order
    replace the plant's color with green
    replace the plant's size with 10
```

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

and deadlock, but they must be able to look at a computing problem and think of solutions that utilize parallel processes.

With the disconnect between sequential-only introductory programming classes and the necessity for programming students to learn parallel programming concepts and methods in mind, we developed an introductory parallel programming course that specifically targeted novice programmers. We designed a language, called *EcoSim*[1] {FIXME Are we going to name the language officially?}, that simulates a parallel programming environment and has a highly accessible natural language syntax. Programs written in *EcoSim* have the ability to exhibit race conditions, allow both atomic and non-atomic variable assignment, and show increased performance when the number of cores is increased. It is a turing-complete language, and contains a number of basic functions geared towards making the programs interesting for novice programmers. 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 2 shows the *EcoSim* development environment, which includes a code window, settings, a console window with output messages, and a window for graphical objects.

We had three overarching goals in mind for the course we designed around *EcoSim*:

1. Introduce the students to simple parallel programming ideas using multiple processors.

2. Provide interesting parallel programming examples the students could easily modify and learn from.

3. Teach the students to "think parallel" about computing problems we gave them, or that they thought up on their own. {FIXME should we include the define the task/describe a solution/tell the computer here?}

We presented the course, titled, "Programming the Computers of the Future" to two classes of eighteen 4th and 5th grade (9 and 10 year old) students during a five-day enrichment program. Each class period was two hours long, and the students had a week between classes, although they could access the programming development environment online to continue learning independently. None of the students had significant prior programming experience. We based the course curriculum on creating a simulated ecosystem, starting with simple objects such as stationary plants that could grow in place, and eventually creating herbivores and carnivores that could move about the screen. Our

[1] *EcoSim* is so-named because the original class we taught with it focused on an *eco*logical *sim*ulation.

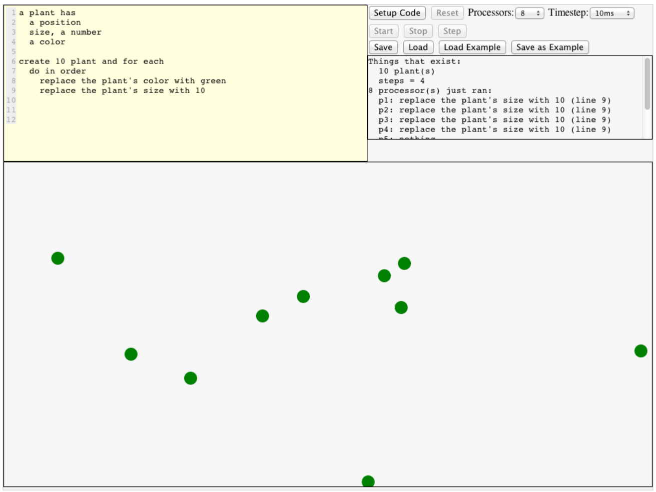

Figure 2: The *EcoSim* web-based integrated development environment hosted at `http://ecosimulation.com`. Code is written and debugged in the top left window, settings are on the top right, a console with runtime and debug information is below the settings, and the main window shows the graphical output of the program.

lessons included group exercises that introduced parallel programming concepts and general programming-style problem solving, and each lesson included example *EcoSim* programs with time for the students to modify or attempt to create new programs on their own.

We had many successes in our pilot course:

1. Exit surveys collected from the students in both classes showed enthusiastic responses to the class, and students reported that they learned a number of programming concepts.

2. Student code examples show that by the end of the class students were familiar with the language and were able to write programs that took advantage of parallel concepts.

3. After one or two classes the students felt comfortable with basic concepts of *EcoSim* and were able to write rudimentary parallel programs without trouble. By the end of the course, a number of students designed and implemented creative programs that highlighted the parallel nature of the language.

2. BACKGROUND AND RELATED WORK

Parallel computing has a long history, dating back to 1955 and the IBM 704 and its ability to compute parallel arithmetic [8]. Amdahl's law, defining the maximum possible speedup due to parallelization, was coined in 1967 [3], and multiprocessor mainframes and multinode distributed computing platforms provided most of the world's parallel processing until the early 2000s. However, even with multiprocessor systems, programmers were first taught how to write sequential applications, generally learning parallel programming concepts for specific computers or platforms. The microcomputer explosion of the late 70s and early 80s ensured that most programmers were exposed to uniprocessor machines as their first computers, and thus their first programming experiences were with sequential programming languages as well. Today, multicore desktop and laptop computers are ubiquitous, and in order to make the most efficient use of these computers, parallel programming is necessary. Furthermore, when novice programmers sit down to write their first code, it is using a a parallel computer.

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 importance than simply learning the sequential aspects of the language. There have been a number of studies on teaching parallel programming concepts using traditional languages at the undergraduate level [9, 12, 14] and at least one at the secondary school level [13].

There are also languages designed for parallel programming, but they tend have advanced syntax and be targeted towards students already proficient at programming in general (e.g., X10[6], NESL [5], and Go [1]). It would be hard to suggest any of these languages to an absolute beginner programmer.

3. ECOSIM: AN INTRODUCTORY PARALLEL PROGRAMMING LANGUAGE

To teach programming, it is necessary to select a language or environment. We had a number of constraints on the language we taught:

- It should be fundamentally parallel.
- The meaning of programs should require little explanation.
- It should run without installation.

Since we were aware of no language having all of these characteristics, we designed one. To make it run without installation, we implemented it entirely in javascript and wrapped it in website having a simple editor and interaction environment. Notably absent from our goals was being a "complete" language; for example, since we did expect to teach pointers, we did not include pointer mechanisms in the language.

3.1 Themes and Mantras of EcoSim

We wanted out language to get out of the way of teaching parallel computing principals and practice. To facilitate this goal, we developed a set of guiding themes or mantras by which we designed the language.

First and most important, we wanted each core statement to be it's own explanation. When considering a possible language syntax, we would ask "How would we explain this construct to a complete beginner? Can we replace it with that description?" For example, we would describe the behavior of the assignment statement `x = 3` as "replacing the value stored in `x` with `3`", so we decided to write is as `replace x with 3`.

Second, we wanted to make sure students didn't loose sight of the fact that some processor must execute each statement. We decided to realize this goal by having statement in EcoSim be written to *address* a processor. Thus, instead of the OO-style `gen.nextInt()` we preferred `get the next number from gen` or `gen's next number`.

Finally, we wanted to make naming values the exception instead of the rule. In common speech we rarely name values; instead we say things like "the square of *a number* is *the number* times *itself*", making use of common reference techniques in English.

3.2 EcoSim Syntax & Semantics

EcoSim was implemented as a statically-typed interpreted language using a modified recursive descent parser that type-checked and parsed in the same pass, allowing some degree of context sensitivity in the language definition. For example, consider the line `how to cow a person` is a syntax error if "cow" is a type, declares a single-parameter subrountine "cow" is "person" is a type, and declares a parameterless subroutine "cow a person" if neither are types. While this parser design allowed us to easily implement multi-word statements, it also proved somewhat difficult to implement correctly. We expect there are still several corner cases we failed to handle correctly.

1. Overview
2. Language semantics
3. Language syntax (natural language)
4. The graphical interface (including collision detection, wall detection, etc.)
5. Things that set it apart from other parallel programming languages
6. Examples

4. COURSE OVERVIEW AND LESSON PLANS

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, and both were targeted for our audience of self-selected primary school students with no prior formal programming experience.

1. Overview (ecosystem in parallel)
2. Starting to "think parallel" – student sort
3. Ingraining the idea of multiple processors working independently to solve the same problem.
4. Constantly let kids show off what they have accomplished
5. the idea of "when bored", for some, for all
6. atomicity and race conditions (class example on board – roll/read/roll/write)
7. St. Matthew Island

4.1 Ecosystem in Parallel

The original conception of the pilot course was, simply, "Let's teach fourth and fifth graders about parallel programming." We decided on an "ecosystem" theme for the course, based on a number of reasons. First, students at this level are familiar with real-life ecosystems, and we felt that they would find the topic interesting. Second, ecosystems have a number of embarrassingly parallel characteristics; for example, in a forest there are multiple copies of trees which can each be handled independently and in parallel. Finally, we knew we could model a simple ecosystem and then build upon the original model to make it more complex. Starting with a forest of stationary plants that have a single "grow" characteristic, we added motile herbivores that consumed the plants. We then added the ability for the inhabitants to reproduce and gave them the ability to die from starvation, and then eventually we added carnivores as well. By the end of the course students had expanded the ecosystem to include plants that only grew during the day, hunters, and even carnivorous and poisonous plants.

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 not only slows the computer to a crawl, but the herbivores quickly decimate the plant population and start to die from starvation. We spent a number of classes discussing and modeling the intriguing real-life case of a herd of reindeer who overpopulated a remote island in Alaska and subsequently died out [10, 11], and with the *EcoSim* model the students could adjust the parameters to find an equilibrium that would have allowed the reindeer to survive.

4.2 Getting the students to "think parallel"

At the beginning of each class period and before writing any code, we first introduced the students to a parallel programming concept in a full-class discussion, usually with an activity. For instance, 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 two at a time. Unsurprisingly, this took over three minutes, and this led to a fruitful discussion on why parallel processing can be faster.

Table 1 shows the group activities we conducted an 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, after the race condition activity, we wrote the programs in Figure **??**, which demonstrate a race condition stemming from allowing multiple processors to complete the `color` statements in any order.

5. STUDENT WORK AND OUTCOMES

6. CONCLUSIONS

Conclusions

In order:

```
a moth has
    a position
    a color

create 10 moth and for each
    do in order
        replace the moth's color with gray
        replace the moth's color with black
```

In any order:

```
a moth has
    a position
    a color

create 10 moth and for each
    do in any order
        replace the moth's color with gray
        replace the moth's color with black
```

Figure 3: 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.

7. REFERENCES

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

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

[3] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[4] 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.

[5] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[6] 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.

[7] 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.

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

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

[10] D. Klein. The introduction, increase, and crash of

Group Activity	Parallel Programming Concept
Students sort themselves, and then one student sorts everyone.	Parallel speedup
Everyone shares a pen to write on the whiteboard to increment a number.	Locks / Atomicity
Students roll a set of dice until they roll a specific combination. Then they look at the board for a number, increment, and call out the new number, which is written on the board.	Race Conditions
All students start with a number, and half hand to their neighbor to add together. This continues until one student has the total sum.	Reduction and Divide/Conquer

Table 1: Group activities.

reindeer on st. matthew island. *The Journal of Wildlife Management*, pages 350–367, 1968.

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

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

[13] 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.

[14] 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.