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

D.3.2 [**Concurrent Programming**]: Language Classicia- tions—*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

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. [9]

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 [3]. The trend towards increasing cores to program on a single machine does not show any signs of abating in the near future [6], and therefore parallel programming skills are going to become increasingly important. Programmers

```
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.

must not only thoroughly understand parallel programming concepts such as race conditions, atomicity, synchronization, 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*:

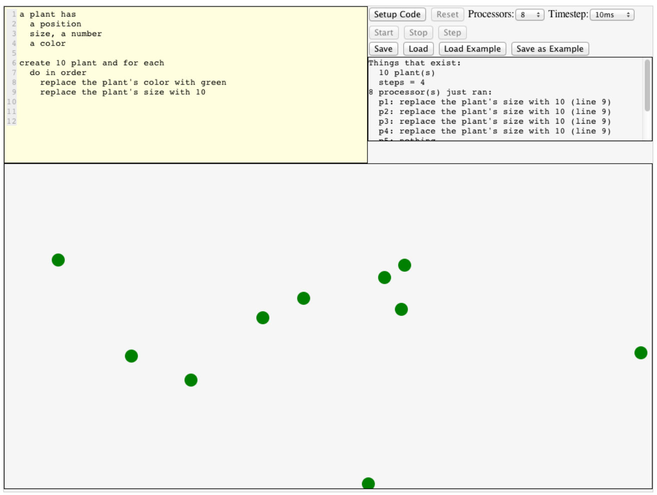
1. Introduce the students to simple parallel programming ideas using multiple processors.

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

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



```
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
```

Processors: 8 Timestep: 10ms

```
Things that exist:
  10 plant(s)
  steps = 4
8 processor(s) just ran:
  p1: replace the plant's size with 10 (line 9)
  p2: replace the plant's size with 10 (line 9)
  p3: replace the plant's size with 10 (line 9)
  p4: replace the plant's size with 10 (line 9)
```

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.

that could grow in place, and eventually creating herbivores and carnivores that could move about the screen. Our 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 [10]. Amdahl's law, defining the maximum possible speedup due to parallelization, was coined in 1967 [4], 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 [11, 16, 19] and at least one at the secondary school level [18].

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[8], NESL [7], and Go [2]){FIXME Should we site ParaSail, CUDA, Cilk, . . . ?}. It would be hard to suggest any of these languages to an absolute beginner programmer.

Several successful languages have been designed to have English-like syntax, including COBOL [1], AppleScript [5], and Wolfram Alpha [13]. Like EcoSim, each was designed to accessible to non-programmers and each targets a relatively narrow set of possible use cases. None, however, seem suited to teaching parallel programming concepts.

Another approach to making "readable" languages is to use a graphical rather than textual representation of the program. Recent successful examples aimed at children include Scratch [14] and Kodu [17]. We do not pretend to be aware of all graphical programming environments but were unaware of any that could be run without installation and used to teaching parallel programming concepts.

3. EcoSim: AN INTRODUCTORY PARALLEL PROGRAMMING LANGUAGE

To teach programming, it is necessary to select a language or environment. We had four characteristics we wanted in the language we taught:

Fundamentally Parallel: We wanted a language that was parallel unless requested otherwise. We also wanted the idea of a processor executing the instructions to be visible throughout the language.

Self-Explanatory: We didn't want to have to translate what code meant. No rebinding "=" to be assignment instead of equality and preferably no new symbols or re-defined words at all.

Web-Based: We wanted the students to be able to work at home without any trouble. This meant using either Javascript or Flash, the only zero-installation tools we could count on *all* the students having.

Engaging and Transparent: We wanted every program the students wrote to be interesting to them, which meant no text outputs. We also wanted a "glass hood" so they could watch the parallel engine running.

Notably absent from these goals is being a "complete" language. Our objective was to teach parallel programming concepts, not necessarily to provide a gateway into serious software development.

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 pure Javascript and created an interaction environment using basic HTML+CSS features and the new HTML5 Canvas element for the graphics.

3.1 The EcoSim Runtime

We determined to structure the EcoSim environment with three basic interfaces: the code entry pane and two visualizations of the program being executed: a status window to listed the behavior of each processor and a graphical display of each object. In the graphical display we decided to automatically draw a circle for each object for which the students' code had defined a position and a size rather than providing some more abstract representation of objects or an explicit graphics API.

In addition to the interfaces, the EcoSim runtime provides the following:

A fixed number of virtual processors. Each processing step the interpreter assigns each processor a task and displays the work performed by the processor in both the code pane and in the status window.

A shared work queue for ongoing operations. 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. Every object that is instantiated is placed on a global list of objects of that type. 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 not collisions to handle then remaining processors are given jobs from a set of low-priority tasks.

To facilitate these operations we provided a built-in notion of only four types: number (IEEE floating points), 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.

3.2 The EcoSim Language

Three mantras guided our design of EcoSim's language constructs: "self-describing syntax", "audience: processor", and "anonymous by default." The first two are direct consequences of our desire to be self-explanatory and to have the idea of a processor visible throughout the language. The last, defaulting to anonymity, was inspired by the hope that anonymity would keep things parallel (if you don't have a name then you are less likely to care which one is present) but proved far more valuable in that it removed the need to spend time trying to think of variable names.

Rather than provide a full description of the language that grew out of these mantras we provide here a few typical examples and rely on the language's self-describing character to render later code examples understandable.

The first operator we considered was the assignment operator. We had found through interactions in CS1 courses that the syntax used in `x = x + 1` caused confusion in many students. 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 that is too verbose and fails the "audience: processor" mantra. We finally settled on "replace x with old x + 1": assignments as "replace *lvalue* with *rvalue*" and the word "old" required for variables in the rvalue that also show up in the lvalue.

A few more examples of this sort: `while(1<2)` becomes "as long as $1 < 2$", `else` becomes "otherwise", `double x = 3` becomes "start x as 3". We replaced the membership operator (commonly `.` or `->`) with the more English-like "'s" as in "baz's position's x" and use type inference to have a statically-typed language without needing to declare variable types.

The runtime keeps a list of objects of each type (see §3.1). We add to these by writing "create 3 number" or

```
1: create 4 number and for each
2:     replace the number with 7
```

We can access just one randomly selected object, or all of them in parallel:

```
1: for some number
2:     destroy the number
3: for each number
4:     replace the number with the old number + 1
```

Structures, properties, and subroutine definitions are describing "what we mean by X" rather than "you should do X"; utilizing the "audience: processor" mantra, these are worded to inform, not direct, the processor.

```
1: a plant has
2:     size, a number
3:     a position
4:     2 color
```

This introduces a structure type named "plant" with a named field "size", an anonymous position field, and two anonymous colors. We can define properties for plants:

```
1: a plant's age is the plant's size - 5
2: 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; you cannot write "the plant's nth color" for variable n.

Subroutines are defined with a "how to":

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

Calling a subroutine is straighforward

```
1: for some plant
2:     add 3 years to the plant
```

It is worth observing that this way of defining and calling subroutines is context sensitive and hence not readily doable with most parser techniques. Our parser type-checks and builds the symbol table in the same pass as it parses, so when we parse a line "how to *words*" we can identify which of the words are identifying types and which ones are 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" with two parameters and not "add years to the plant" with only one.

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

```
1: when a bulldozer hits a plant
2:     destroy the plant
3: when bored
4:     create a plant
```

Again, these are worded to address the processor in a self-explaining way. They prevent the need for a "main" method, since "when bored" will suffice, and make event-oriented programming quite direct. Multiple "when bored" declarations were permitted, with the runtime selecting between them at random.

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.

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 sub-

In order:

```
1: a moth has
2:    a position
3:    a color
4:
5: a moth's size is 50
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:
5: a moth's size is 50
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 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.

sequently died out [12, 15], 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. Each student sat 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 3, which demonstrate a race condition stemming from allowing multiple processors to complete the `color` statements in any order.

EcoSim allows a programmer to set the number of processors that will be used to run the program. We used this to demonstrate a number of concepts to the students, including demonstrating parallel speed-up as well as race conditions. For example, if *EcoSim* is set to use a single processor, and the "In order" program from Figure 3 is amended to remove line 8, the students can see the individual "moths" changing color one at a time. If we change the simulator to run with two processors, the students can easily see that two moths change color at a time, and with 16 processors they can see that all of the moths change all at once and the program completes almost instantly. With *EcoSim*'s ability to step through a program, individual processor activity can be made even more apparent.

4.3 The Use of Example Programs

As with any programming course, example programs played an important role in teaching our course. This was the first time most of the students had seen any programming language at all, and therefore 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 example programs that the instructors can update at any time. Many times during class we would have students pay attention to the projector as we typed in the code for a program, and then we would have them load the example instead of typing it out. This 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, and students that completed assignments before others in the class were able to modify the example programs or load other 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, and we would also keep versions of certain programs so they could see the steps used to create more robust programs.

4.4 Student Assessment

Because this 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, and we displayed their work on the projector and let them describe and run the programs. This motivated the students to create interesting programs, and the students enjoyed showing off their work to 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 building 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.

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.

5. STUDENT WORK AND OUTCOMES

6. CONCLUSIONS

Conclusions

7. REFERENCES

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

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

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

[4] 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.

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

[6] 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.

[7] 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.

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

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

[12] D. Klein. The introduction, increase, and crash of reindeer on st. matthew island. *The Journal of Wildlife Management*, pages 350–367, 1968.

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

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

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

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

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

[18] 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.

[19] 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.