Learning Parallel JavaScript with a Visual Boids Simulation

Kyle Barton, Samuel Dodson, Danielle Fenske, Ben Whitehead & Jens Mache

{kjbarton, sdodson, dfenske, benw, jmache}@lclark.edu

Department of Mathematical Sciences
Lewis & Clark College, Portland, OR, USA

Abstract

As multi-core processors become the standard of modern computing, it is important that computer science students learn parallel programming. Traditional parallel computing education has concentrated on computational science and on low-level techniques like threads or MPI. We expand the parallel computing curriculum [3, 4, 9] by introducing a parallel graphical simulation of flocking birds in JavaScript. As undergraduates, we assess the effectiveness of JavaScript and River Trail as an introduction to parallel computing.

Keywords boids, computer science education, parallel computing, JavaScript, River Trail

1. Introduction

Most modern devices contain multiple processors, and parallel programming is very common among most applications on these devices. This method of programming harnesses the power of multiple processors to speed up performance. However, the Internet is one domain of computing that has not taken full advantage of parallelism, due to a few external factors, such as a lack of a good parallel model. Web applications would greatly benefit from parallelization, especially in the context of 3D animation, physics simulations and video processing.

We explore the possibilities of parallelizing Web applications with Intel's parallel programming API extension to JavaScript, River Trail, to parallelize a flocking simulation created by Craig Reynolds, called boids [8]. River Trail was created by Stephen Herhut at Intel [5], and was presented for the first time at the Intel Developer forum in September 2011. Currently, it exists as a prototype extension to Firefox 21. We chose to explore River Trail in the context of a boids simulation because it is a good way to visualize the speedup due to parallelization. This visual evidence is useful when teaching parallelism, as a way to motivate students and show the benefits of learning to program in parallel[6, 9].

2. JavaScript and Parallel Computing

The boids simulation was programmed in the JavaScript programming language. JavaScript is the scripting language most commonly used on the Internet. The syntax is similar to C and Java, so students should be able to quickly learn how to code in it. Its multi-paradigm design allows the programmer to select a functional or object-oriented programming style.

River Trail consists of one new data type, `ParallelArray`, and six higher-order functions (`combine`, `filter`, `map`, `reduce`, `scan`, and `scatter`) used to manipulate `ParallelArray` [1]. These higher-order functions act as any functional programmer would expect. The functions are carried out in parallel, without the programmer having to specify low-level settings, like the number of threads or order of execution. This abstraction makes the language readily accessible to new parallel programmers.

`ParallelArray`s are immutable, so in order to manipulate them, you must create new (or "freshly minted") `ParallelArray`s based on the existing ones in order to change the data. Although they may be multidimensional, `ParallelArray`s must be rectangular (or uniform) – meaning each row has the same number of columns. Every `ParallelArray` has a shape which describes the dimensionality and size of the array. For example a 4x5 `ParallelArray` has the shape [4,5]. To parallelize a program, the programmer must define a `ParallelArray` and the data it contains, the method that will determine the parallel processing pattern, and an elemental function that is applied to the data in parallel.

The function we were able to parallelize was the `draw` function which is called every `step`. Instead of sequentially going through every boid to draw each according to their position, we were able to draw the boids in parallel. Here is the code we wrote:

```
var boids = new ParallelArray(boids);
var draw = boids.map(function draw(boid) { ... });
```

We create the `ParallelArray` of boids to manipulate, and then call the anonymous function draw on each boid that is executed in parallel. We also attempted to parallelize the `step` function which carries out the flocking algorithms. The `combine` function is well-suited for this, but due to time constraints and an error message stating "`combine is not a function`," we were unable to implement this method.

3. Boids Simulation

The Boids simulation was first created by Craig Reynolds in 1986. It simulates the basic flocking motion of animals such as birds [8].There are three central rules which guide the boids in their trajectories about the screen. Each rule outputs a vector in a certain direction, and after each rule is calculated, the vectors are summed and the boid is moved in the direction of the summed vector. Here are the three rules:

1. **Cohesion** (moving towards the center of mass): This rule states that boids tend towards the average position of all other boids in a neighboring vicinity. It can be implemented simply by adding up all the positions of the neighboring boids and dividing by

the number of boids. This average position is then added to the boids current position, and a fraction of that position is taken so that it moves only slightly towards the center of mass.

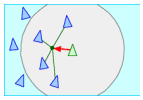

Figure 1. Cohesion

2. **Separation** (making sure that no two boids are too close to each other): The boids must steer to avoid collisions with other boids nearby. This can be implemented by initializing a vector to zero and then subtracting the displacement of each nearby boid from the vector. This results in any two boids that are too close being repelled in the opposite direction from each other until they are far enough away.

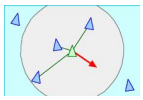

Figure 2. Separation

3. **Alignment** (steering boids towards the average direction and speed of surrounding boids): This rule is calculated by finding the perceived average velocity of surrounding boids and then adding a fraction of that velocity to the specific boid. This ensures that the boid begins to match surrounding velocities, but does not instantly change velocities, causing smooth motion.

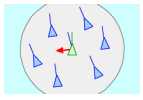

Figure 3. Alignment

There are many other factors one can implement, such as the effect of strong winds, what boids do when they scatter, the behavior of the boids when there is a limit to their speed or their bounds, etc. The final step in the program is independent of how many factors are involved. All one must do is add up all the vectors that result from the rules and adjust the boids current position and velocity accordingly.

The reason this is an embarrassingly parallel program is because at each time step, we can calculate the rules for each boid independently of the others. In an example where there are thousands of boids, instead of sequentially traversing the boids to calculate their new positions, we can use threads to calculate the rules for each boid in parallel. This can cause a significant amount of speedup if done correctly, and the students would be able to observe the effects of parallelization.

An advantage of the boids simulation is that it is graphical. This allows the programmer to see what progress she is making on the simulation. For example, one problem we encountered was that the boids were flocking to the top-left of the screen. Seeing this, we were able to see the bug in our program and correct it in our source file.

4. Discussion

River Trail is not the only JavaScript parallel API. We also explored web workers (another JavaScript parallel API)[7], but we decided against using it because it is much more low-level, and it involves message passing. The programmer must write explicit commands and think at a lower level to facilitate communication between the web workers. Furthermore, each web worker must be in charge of its own JavaScript file, which would not be easy for the boids program. These are the main reasons why River Trail is a better option for beginning programmers. The choice between River Trail and web workers is a tradeoff between simplicity and flexibility.

There are many ways to improve this simulation, and numerous unexplored paths to follow. The boids simulation has been implemented with a variety of additions, including 3D effects, predator and prey models, and other biological phenomenons. All of these simulations could be further improved with parallelization. Due to certain time constraints, we were un able to analyze performance of our program and compare with the sequential version, but a performance comparison would be very valuable to this project as well. Furthermore, other parallel APIs do exist for JavaScript, such as web workers. It would be interesting to explore the similarities and differences between the two parallel APIs.

We are also creating a tutorial for undergraduate education using the boids simulation. In short, the tutorial will provide the sequential boids code and, after hearing an explanation of boids, the students will be instructed to parallelize as much of the code as possible. This is educational because there are many opportunities for parallelization in the code. This tutorial would fit within a Networks or Web Programming course, where JavaScript is already part of the coursework. This allows the instructor to include a short lesson on the possibilities for parallelization, without having to alter his or her lesson plan too drastically. The Boids simulation is a great demo for modeling the benefits of parallelization. We are also exploring how best to create a teaching module with boids to post to the National Science Foundation supported CSinParallel[2].

Acknowledgments

Partial support for this work was provided by the National Science Foundation award number 1044932, by the John S. Rogers Science Research Program, and by the James F. and Marion L. Miller Foundation. We would also like to thank Professors Richard Brown and David Bunde.

References

[1] `http://rivertrail.github.io/RiverTrail/tutorial/`

[2] `http://serc.carleton.edu/csinparallel/index.html`

[3] Bunde and Mache. *Teahing Concurrency Beyond HPC*. In *proceedings of the 2009 Workshop on Curricula in Concurrency and Parallelism, OOPSLA 2009*.

[4] David Bunde, Jens Mache, Casey Samoore, Sung Joo Lee, Jonathan Ebbers, Christopher T. Mitchell, Julian H. Dale. *Teaching Parallel Computing with Highe-Level Languages and Compelling Examples*. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE), 2012*.

[5] Herhut, Stephan, Hudson, Richard L., Shpeisman, Tatiana and Sreeram, Jaswanth. *Parallel Programming for the Web*. Berkeley, CA. In *HotPar'12*.

[6] Mitchell, Christopher T., Mache, Jens and Karavanic, Karen L. *Learning CUDA: lab exercises and experiences, part 2*. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications*, 2011.

[7] Mozilla Developer Network. *Web Workers*. 17 May, 2013.

[8] Reynolds, Craig W. *Flocks, Herds, and Schools: A Distributed Behavioral Model*. In *Computer Graphics*, 21(4), July 1987, pp. 25-34. (ACM SIGGRAPH '87 Conference Proceedings, 1987.)

[9] Sreeram, Jaswanth; Stephan Herhut; Richard L. Hudson; Tatiana Shpeisman. *Teaching Parallelism with River Trail*. Intel Labs.