Data Visualization in Julia

Andres Lopez-Pineda
December 13th, 2011

Introduction:

I am Course 6 and 18, and I am primarily interested in Human Interfaces and Graphics. I took this class because I believe that Parallel Computing is on the cusp of wide-use and the current tools have terrible visualizations, making it much harder for people new to the subject to approach the topics. Thus I wanted to work on visualization of data communications in Julia, which is a new programming language made to be extremely easy to use and approachable for people new to technical computing and parallel computing. With the work already being done in the web interface, I think Julia is looking great as a tool for the average person to use to get their work done easily and quickly see the results with their own eyes. I hope to bring some of that ease of understanding to the development of parallel algorithms. In this paper, I will go through each step in proper UI testing: first, I will describe the setup and tasks that will be completed, as well as the goals for the project. Next, I will enumerate the different types of users being tested. Then, I will describe what has been done previously for other platforms, and several iterations of my solution. Finally, I will show the results of my testing and what analysis I have drawn from the data.

Problem Statement:

Parallel computing is something many scientists haven't thought about, but it would greatly increase efficiency for many of their programs. Some technical computing languages (Matlab, etc) have parallel implementations (Parallel Computing Toolbox for Matlab is one example), but Julia, a new language spearheaded by Alan Edelman, approaches it from a higher level, and with parallel computing in mind from the start. It makes parallel computing simple and easy to add to a preexisting project. However, many people who would benefit from this have no idea how to correctly write parallel code (which is why they aren't already doing it). They will write a piece of code, and even if they have many processors available, they won't take advantage of them. Even if they do add parallelism to their code, they often won't know why their code doesn't see a proportionate speedup (they might even see a slowdown sometimes!). The task of this visualization is to show them exactly how data is being transferred in the hope that this will allow them to get a better idea of where the bottlenecks in their algorithm are and to remove those bottlenecks. The user will run their algorithm, and our modified version of Julia will log data about how they are transferring data. The user will then run the visualization back in real time to see how data was flowing while it ran, then they will go back to the code, modify it to remove bottlenecks, and rerun, once again checking the visualization and (hopefully) seeing a vast improvement.

Task Analysis:

The task analysis includes the 3 different tasks I expect users to do while working on parallel Julia code, so I include them here as what tasks I hope to assist with. These are tasks that I asked my users to do for my tests.

Watch the visualization playback

- Goal: Find bottlenecks in the algorithm by watching the data transfer between nodes
- Precondition: The user has designed and run the algorithm, and the data is available in the visualization
- Subtasks:
  - Monitor the data being transferred between nodes
  - Compare the visual information with the corresponding areas in code where communication is

Improve code

- Goal: Remove bottleneck in the algorithm
- Precondition: The user has seen the visualization and recognized the bottleneck in the code
- Subtasks:
  - Find the place in code where the bottleneck is
  - Understood problem with current implementation
  - Found way to improve based on Julia documentation
  - Changed code appropriately

Confirm Improvement

- Goal: Confirm bottleneck no longer exists in algorithm
- Precondition: The user believes he/she has correctly identified the bottleneck and made the appropriate changes to correct it
- Subtasks:
  - Rerun the algorithm (simulated)
  - See new visualization
  - Compare to old visualization
  - Confirm (or deny) improvement

User Analysis:

I focused on two user classes in my tests, those who have seen Julia before but not parallel computing, and those who haven't seen either before. Each user in my test fell into one of these two buckets, since I found users who were new to both, then gave half of them a day to look at documentation and some Julia examples, fitting them into the second class.

New User to Julia and Parallel
- Motivation: Interested in trying out Julia, has used Matlab and other technical computing languages before for analysis of data
- Feels confident using scientific computing, and has worked on algorithmic problems like this before

New User to Parallel Computing, has used Julia
- Motivation: Has seen Julia documentation and examples, now wants to try out the parallel aspects of it
- Feels confident using scientific computing, and has worked on algorithmic problems like this before

Scenarios:

I chose three problems that I felt were representative of the problems new users would have to deal with when beginning to use parallel computing in Julia. They are contrived and trivial in most cases, but I believe they are still a good way to show some basic problems in parallel algorithms.

Easy Problem
- Setup: Embarrassingly parallel problem, very simple code
- Problem: Starts out with a simple for loop, need to change to a parallel for loop
- Visual: The visualization shows no data being transferred and long runtime

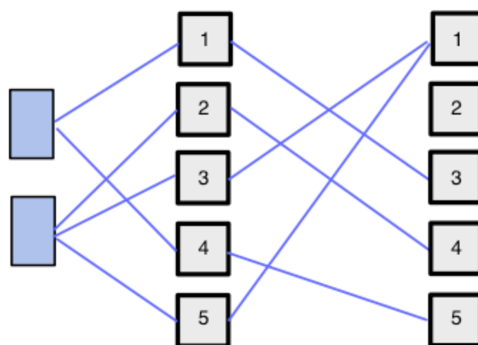
Medium Problem
- Setup: Code farms out some hard operations to other processors
- Problem: Farms out all code to the same processor, should be sending to others
- Visual: Shows all data going to one processor

Hard Problem
- Setup: The code asks other processors to do some operations, then puts the results in an array and does operations on sections of that array
- Problem: phase one sends data out, then the nodes send data back and first node does all work locally, should use a DArray then farm out the operation also
- Visual: Shows data going to nodes, data coming back, then long time on last node

Iterations:

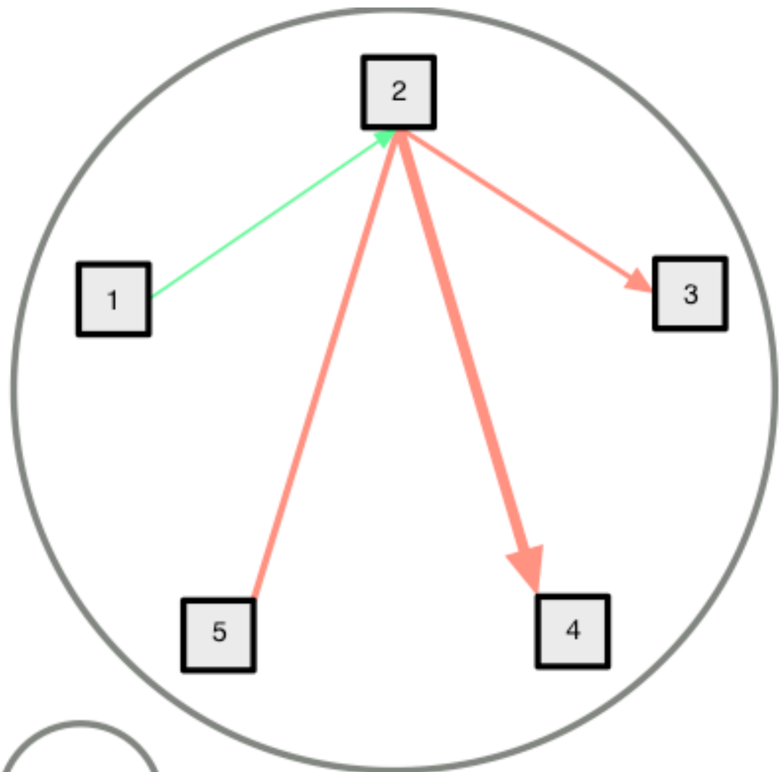
I created a first iteration in Artboard, then animated it using Hype (which creates HTML5 and Javascript code). I then tested it with my users, analyzed the results and iterated on my design again. I show both of their static designs below, with some comments. See the next section for the results supporting the conclusions I draw here.

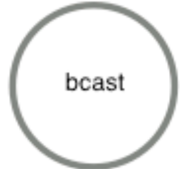
First Iteration:
My first iteration just tried to get across the idea of processors transferring data and working on DArrays. I had two columns of nodes, with the left column being the node originating the data transfer and the right column receiving the data. The DArrays were listed to the left of the nodes.



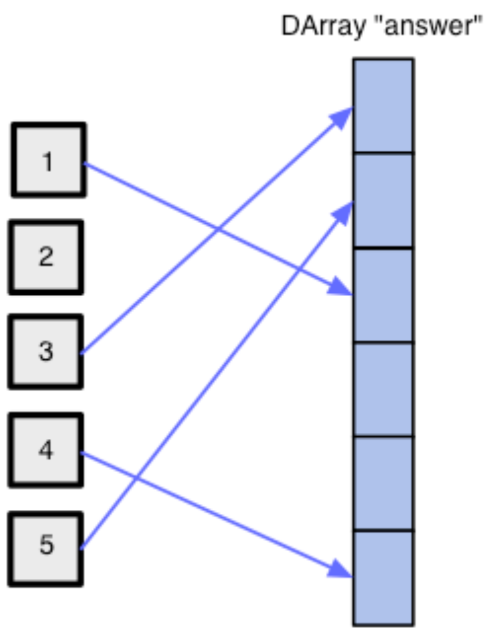
The main problem was that it wasn't clear which processor was receiving the data and which was sending it. It also wasn't clear what has happening with the DArrays. Most users had trouble deciphering the animation because of this, and asked a lot of questions. As you can see in the results section following this one, the users didn't do that well with this visualization.

Second Iteration:
For my second iteration, I moved from a linear approach to a circular approach, and added directionality to my message arrows. This allows me to get across more information with less shapes. I also put in much more contextual information, such as a legend and using different colors for different types of messages. This helped users greatly. However, there were still more questions asked of me then I hoped for. I think that having a tutorial on parallel computing would help; many of them thought the problems were stemming from Julia specific problems, not an incorrect usage of their nodes. Users also requested some way of showing which of the processors were doing work, so if I were to make another design, I would include something showing that.





Line Weights Indicate Amount of Data



Results:

I had 30 volunteers, each of which have done some amount of technical computing (usually Matlab), and gave 15 of them a day to browse over the Julia documentation and see some example code before going through my tests. Each user would be asked to inspect a piece of code and was asked for ideas on it could be improved. At this point, the test went one of two ways. If the user correctly identified the problem with the code, they were shown the best result. Otherwise, they were shown the bad result, with suboptimal runtime. They were then asked if the result was as expected or if it was still subpar. This was repeated for three different problems, of different difficulties. Note that the label “Before” refers to the users’ confidence before making any changes, but after they’ve inspected the code and a visualization (if applicable). The label “Bad Result” refers to the users’ confidence after suggesting a change to code that isn’t correct. The label “Good Result” refers to the users’ confidence after suggesting the correct change to the code.

The users were split into three groups. The control group only saw the code and runtime before asking for ideas for improvements. The other two groups got to see one of the two visualization before and after correcting the code to assist in determining both what is wrong and whether or not the corrections worked.

To quantify the results, I asked each user to ask how confident they were in the runtime of their code and how confident they felt they understood the way the algorithm works, both out of 10. I show the tables of data below, followed by graphs showing the data visually. Afterwards I analyze trends I saw and what they mean for my visualizations.

Easy Problem:

		Control (10 ea.)			Iteration 1 (10 ea.)			Iteration 2 (10 ea.)		
Prior Knowledge	**Average of Conf.**	Before	Bad Result	Good Result	Before	Bad Result	Good Result	Before	Bad Result	Good Result
New to Both (15)	# Users	5	2	3	5	3	2	5	3	2
	Runtime	2.0	3.8	4.4	2.7	4.3	6.7	3.2	4.6	7.2
	Algorithm	7.2	7.5	7.1	5.8	7.6	6.9	5.5	8.1	6.4
Julia User (15)	Total	5	1	4	5	0	5	5	1	4
	Runtime	2.4	3.2	4.6	1.3	N/A	8.9	2.2	5.1	8.1
	Algorithm	8.7	9.2	8.4	8.4	N/A	9.2	9.1	8.8	9.5

Medium Problem:

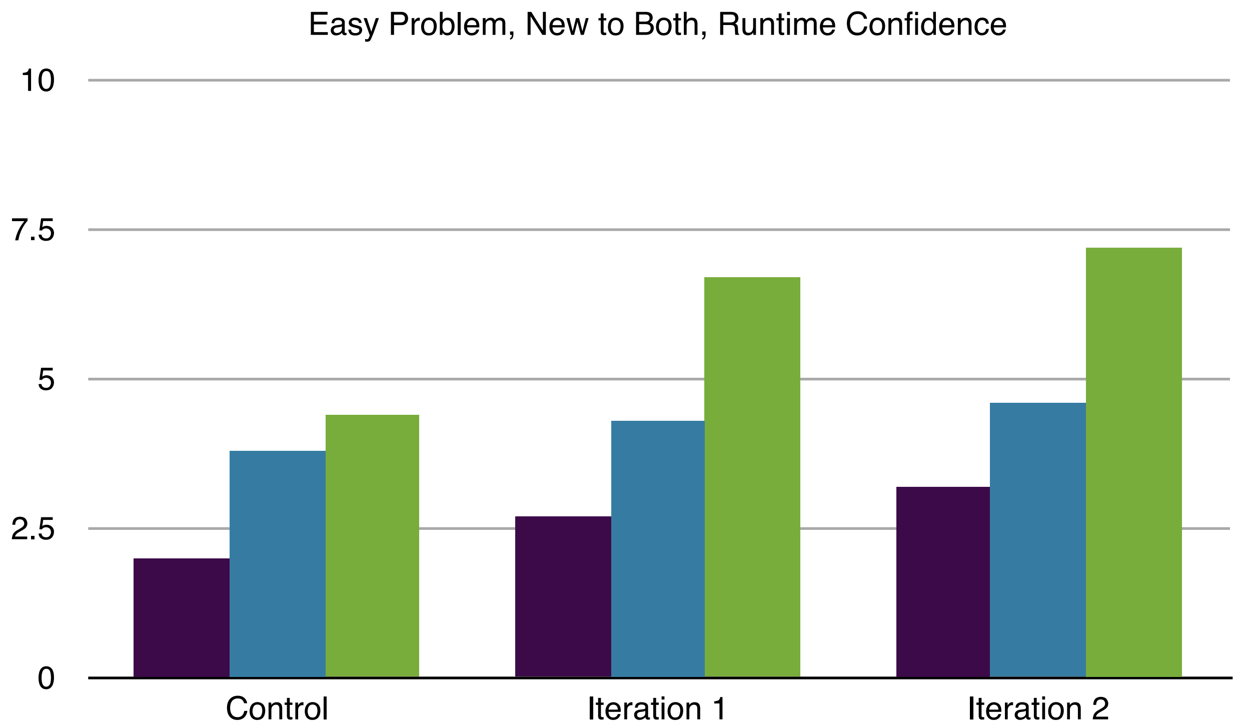
Prior Knowledge	Average of Conf.	Control (10 ea.)			Iteration 1 (10 ea.)			Iteration 2 (10 ea.)		
		Before	Bad Result	Good Result	Before	Bad Result	Good Result	Before	Bad Result	Good Result
New to Both (15)	# Users	5	2	3	5	1	4	5	3	2
	Runtime	1.6	4.0	3.5	3.0	4.1	6.7	4.5	3.4	6.7
	Algorithm	6.9	7.3	8.2	5.5	7.1	6.6	5.7	8.3	6.8
Julia User (15)	Total	5	1	4	5	0	5	5	2	3
	Runtime	2.8	3.5	5.8	2.0	N/A	8.3	1.8	4.2	8.6
	Algorithm	9.0	8.7	8.9	7.2	N/A	8.6	8.9	9.3	9.7

Hard Problem:

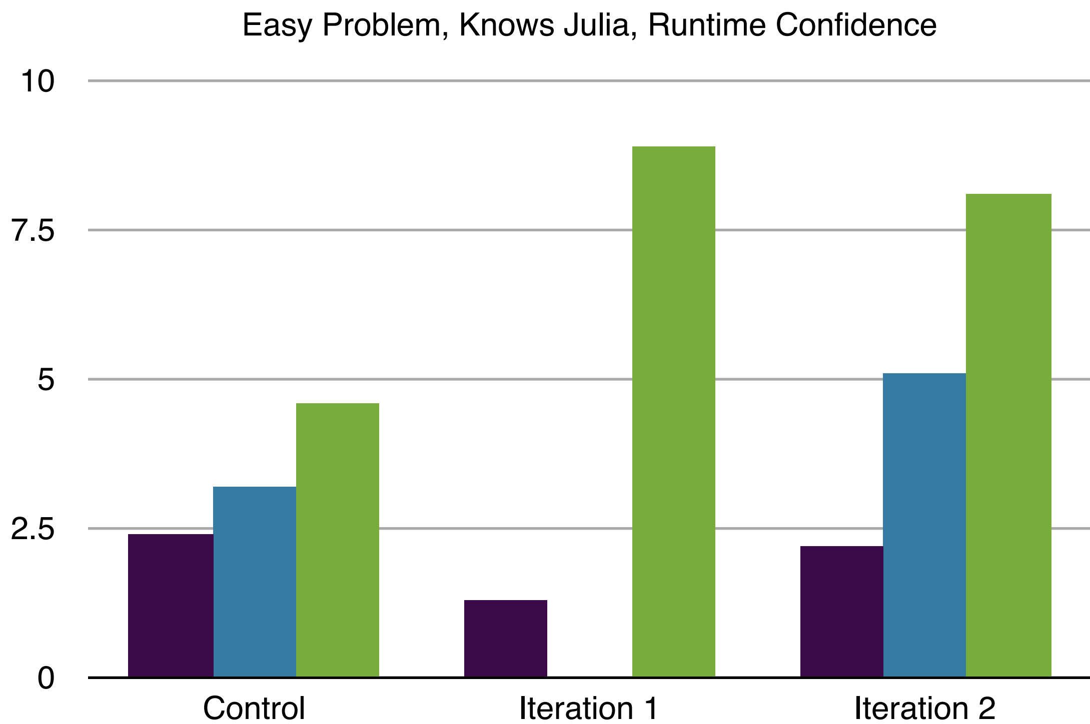
Prior Knowledge	Average of Conf.	Control (10 ea.)			Iteration 1 (10 ea.)			Iteration 2 (10 ea.)		
		Before	Bad Result	Good Result	Before	Bad Result	Good Result	Before	Bad Result	Good Result
New to Both (15)	# Users	5	4	1	5	1	4	5	2	3
	Runtime	4.0	3.2	3.7	2.3	4.1	6.0	3.5	4.4	7.9
	Algorithm	5.8	6.3	6.4	4.9	6.2	6.0	5.1	7.7	6.9
Julia User (15)	Total	5	3	2	5	3	2	5	2	3
	Runtime	3.0	4.1	3.8	1.9	5.0	7.4	3.4	6.8	7.1
	Algorithm	6.0	6.4	6.8	7.1	7.4	7.5	7.3	7.5	7.4

Easy Problem, Runtime Confidence Graphs:

Before Bad Result Good Result



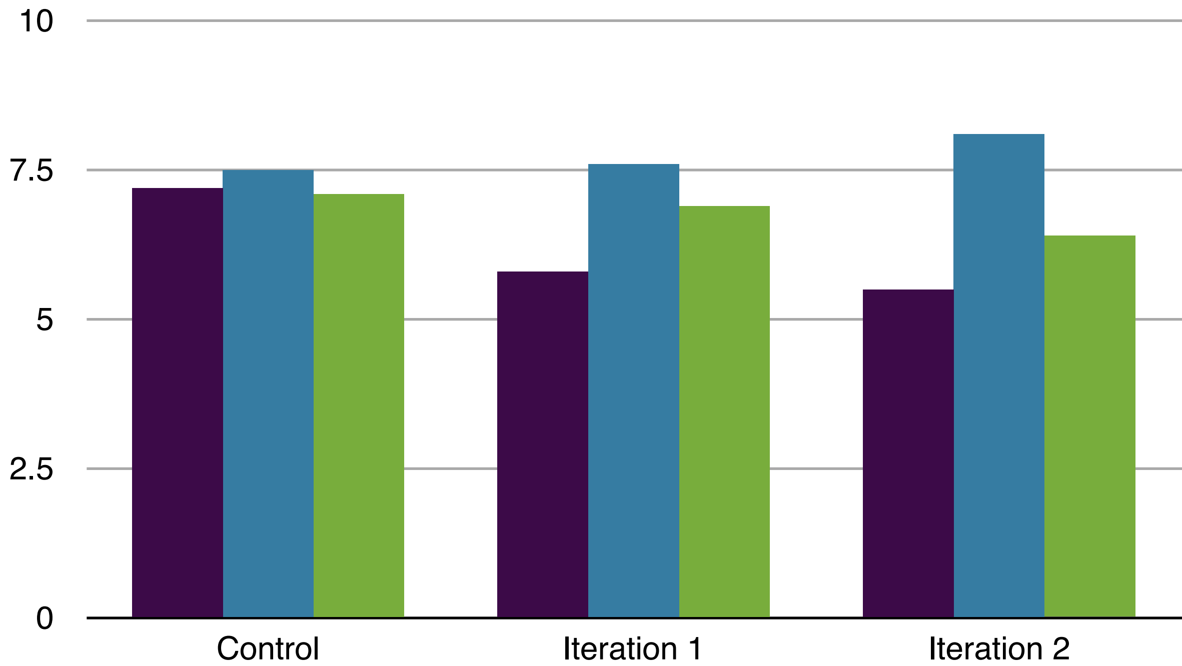
Before Bad Result Good Result



Easy Problem, Algorithm Understanding Graph:

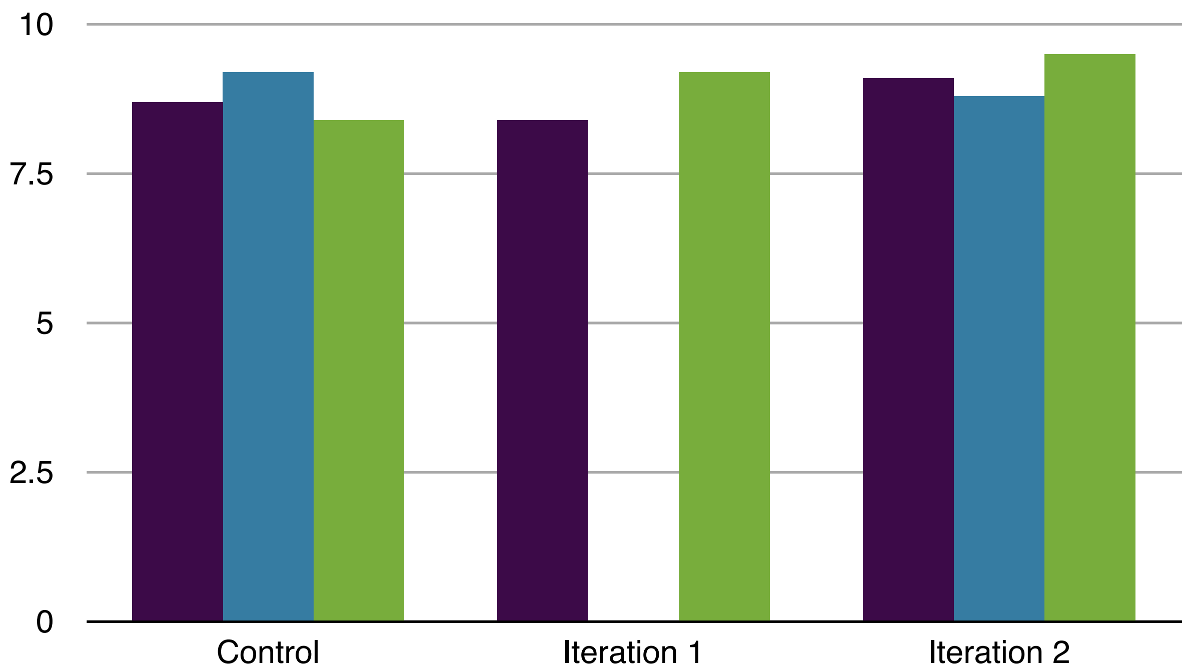
Before Bad Result Good Result

Easy Problem, New to Both, Algorithm Understanding

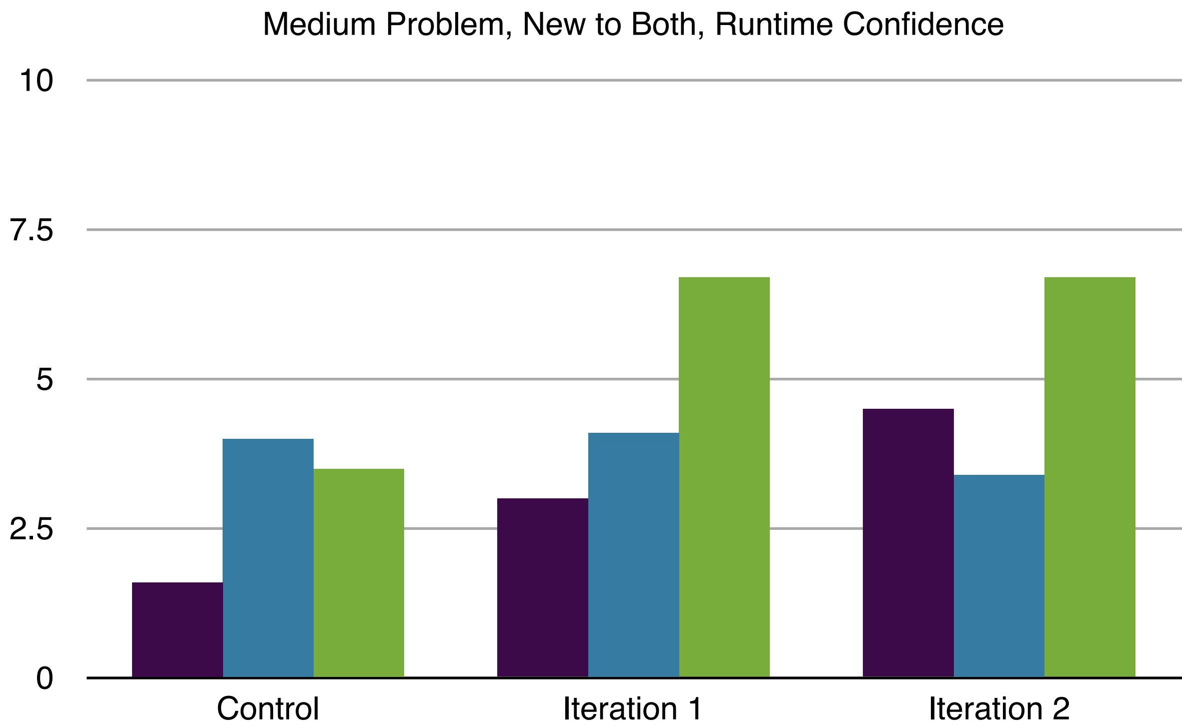


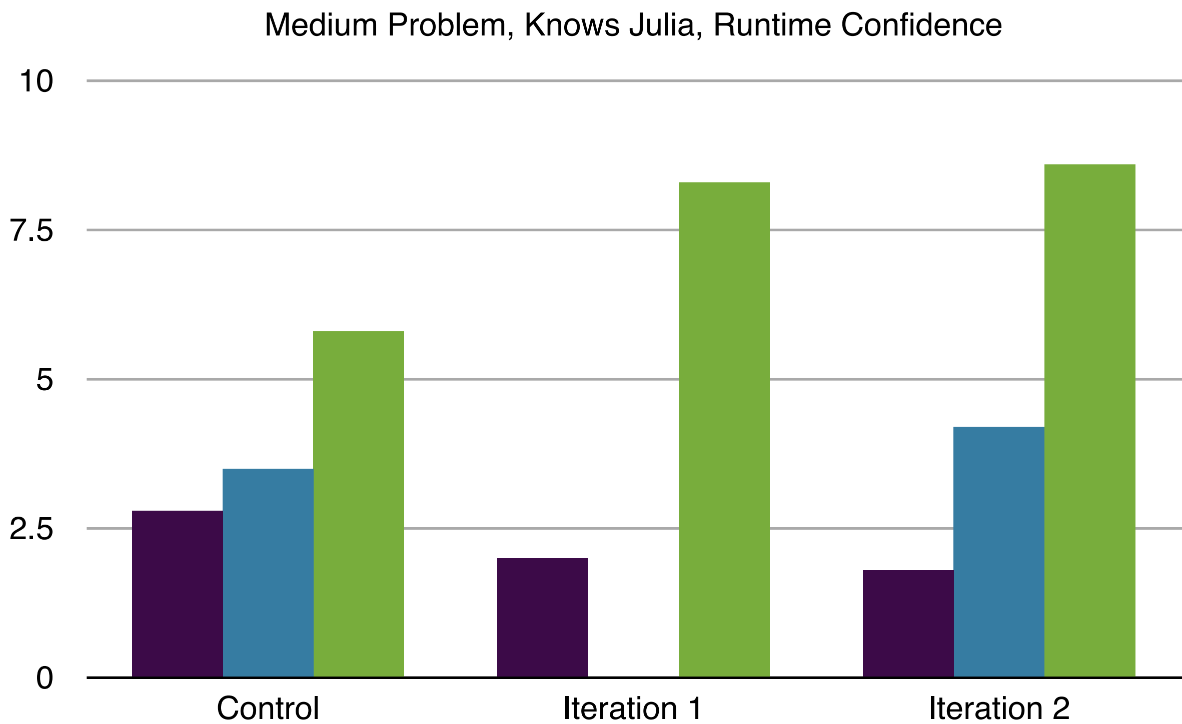
Before Bad Result Good Result

Easy Problem, Knows Julia, Algorithm Understanding

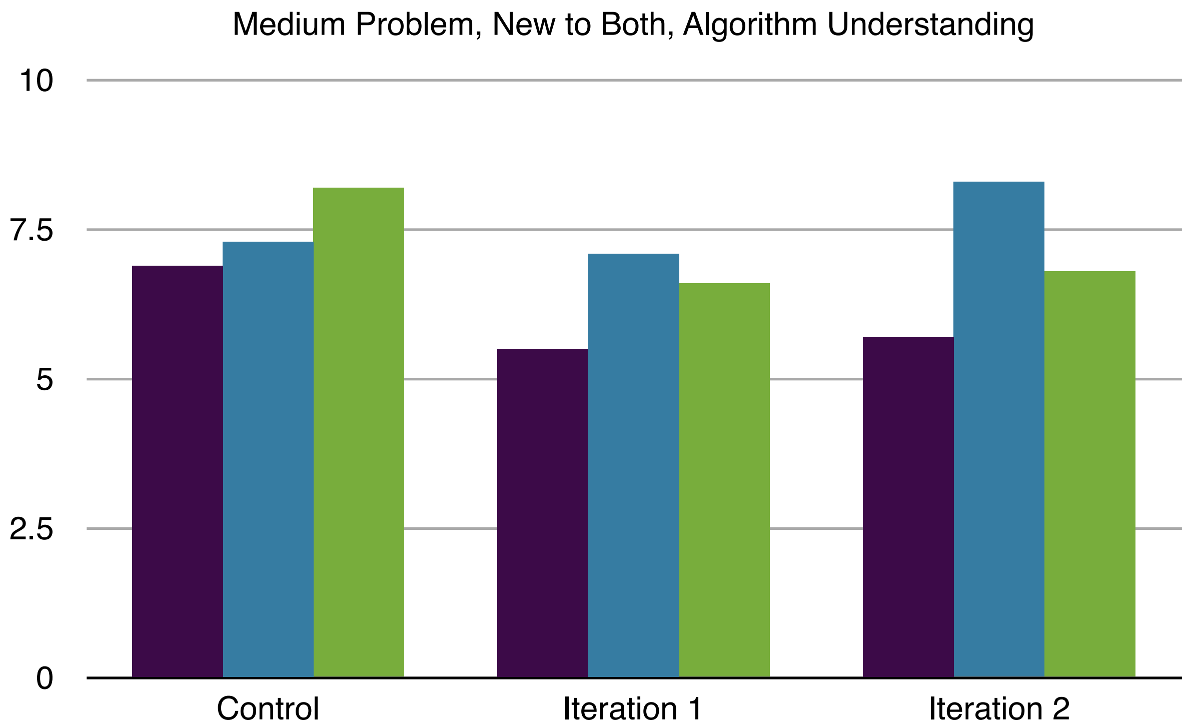


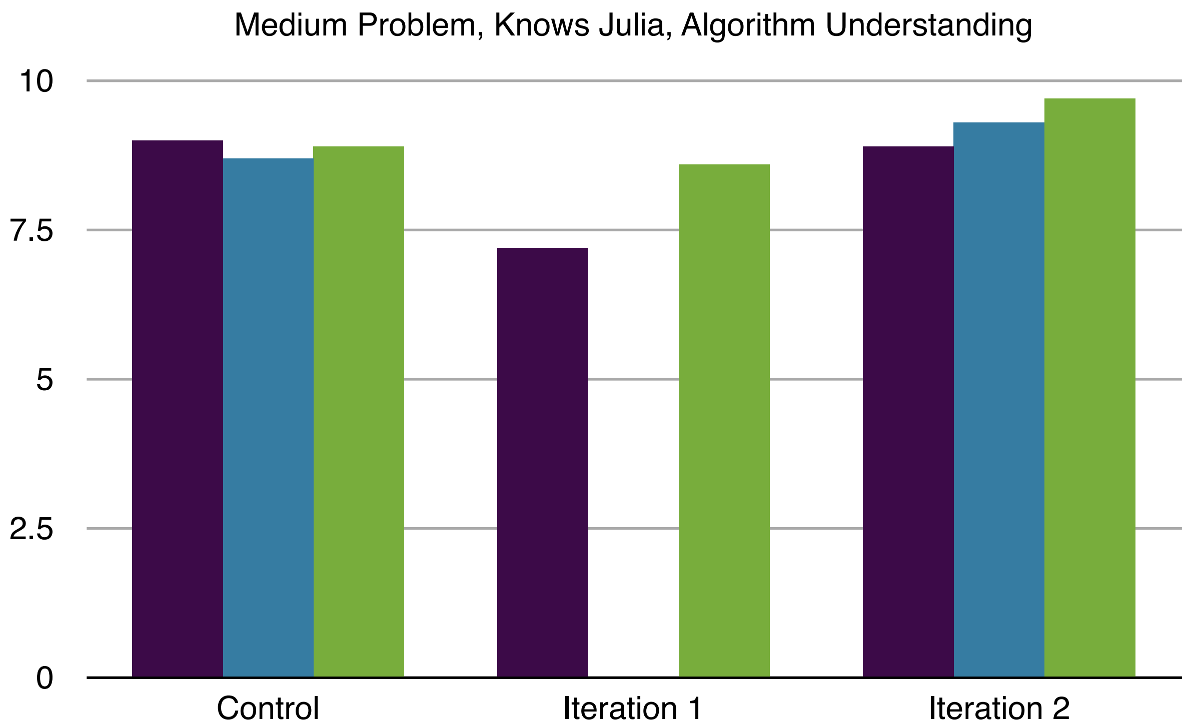
Medium Problem, Runtime Confidence Graphs:



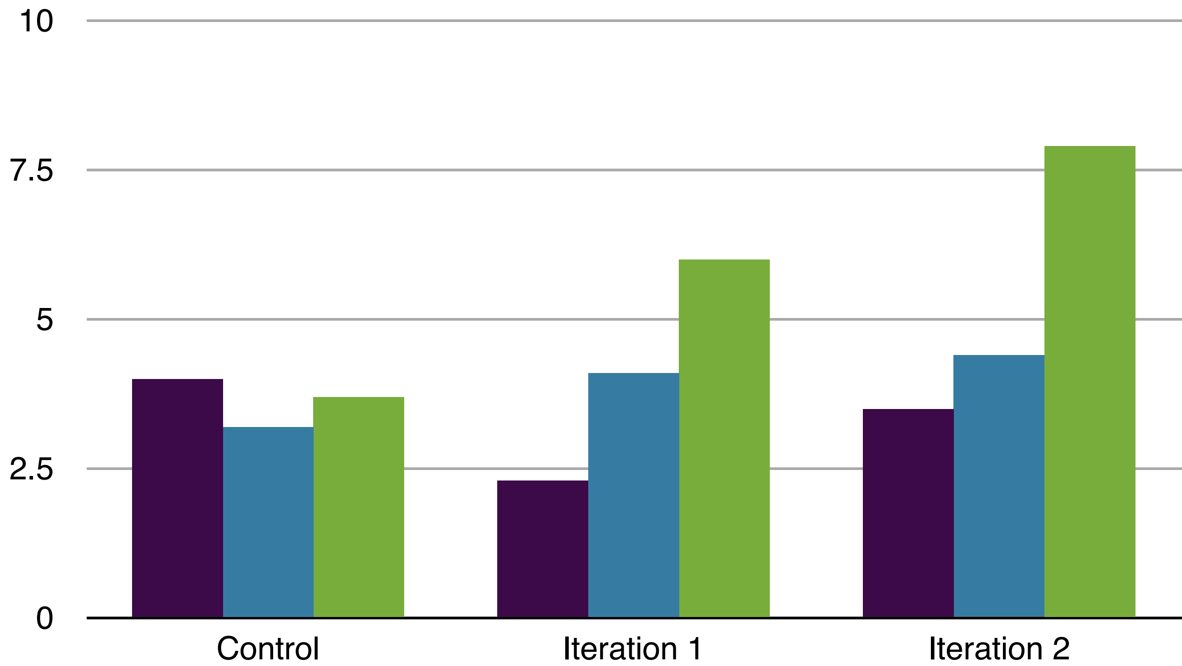


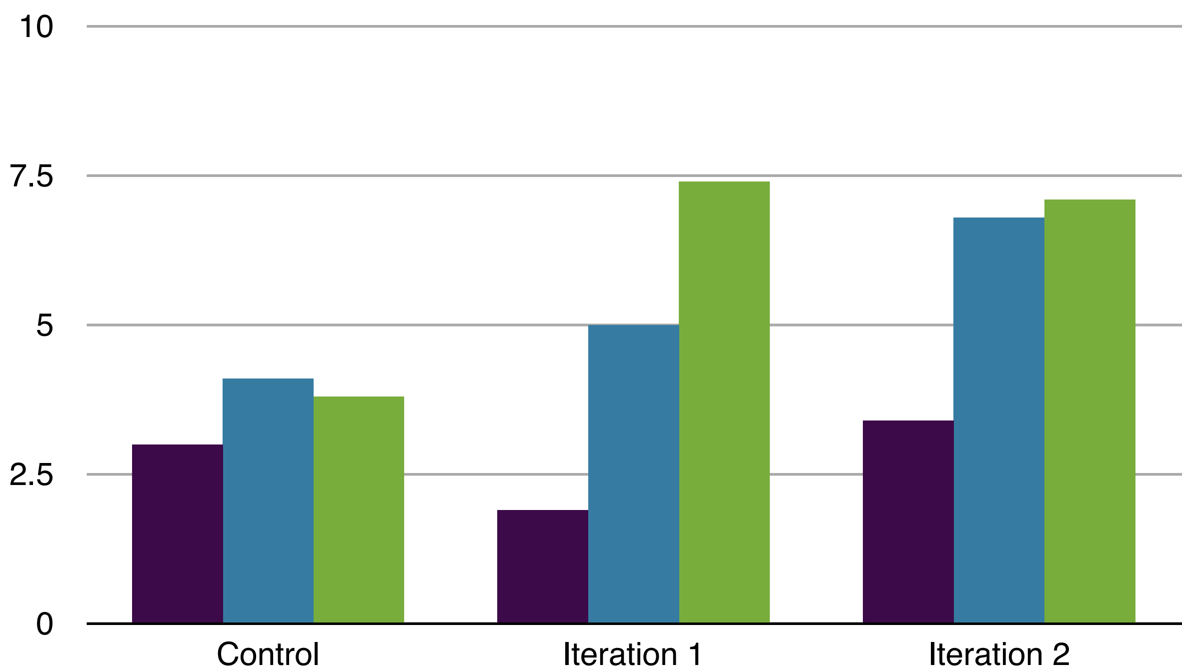
Medium Problem, Algorithm Understanding Graphs:





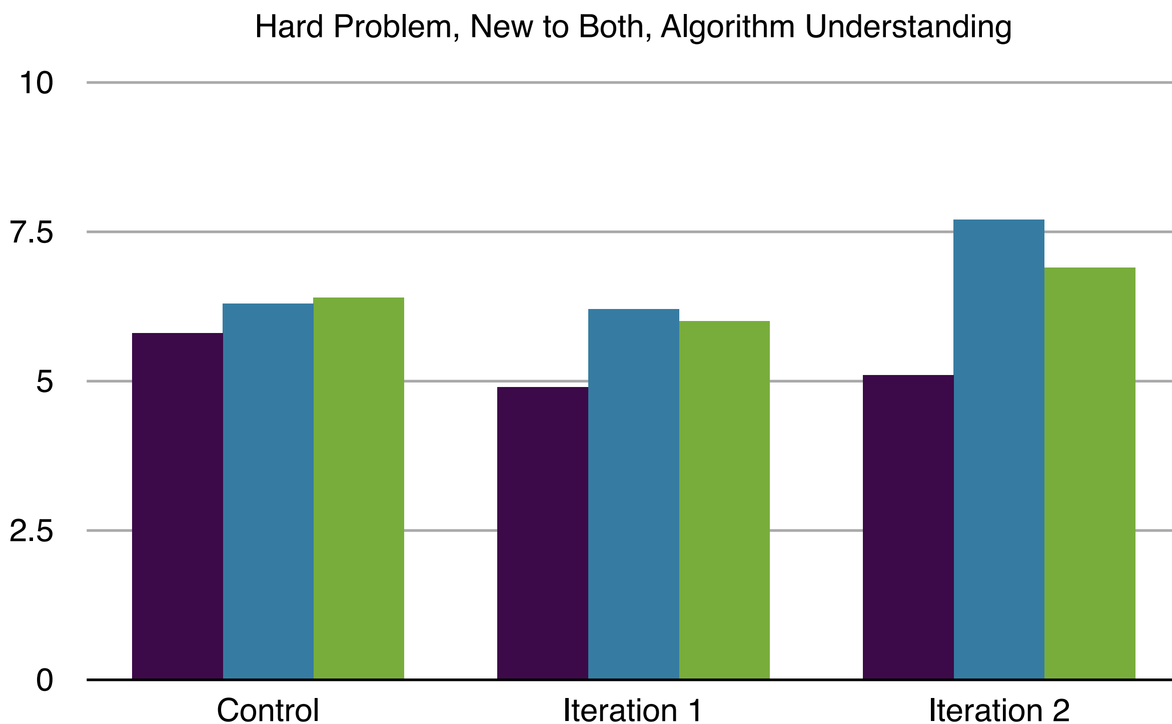
Hard Problem, Runtime Confidence Graphs:

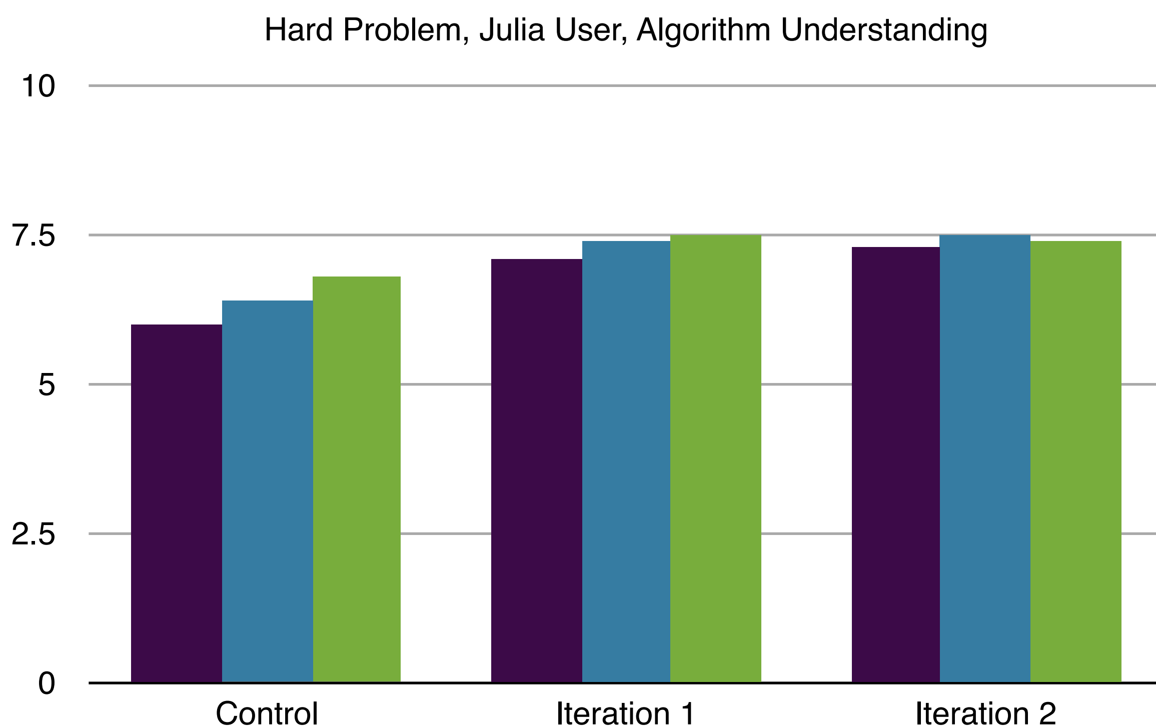




Hard Problem, Algorithm Understanding Graphs:

■ Before ■ Bad Result ■ Good Result





Analysis:

I was pleased with the results overall. Algorithm confidence increased from control to first iteration to the second iteration, which means that the visualization helped them understand what the algorithm was doing and how data moved around between the processors. The runtime confidence is higher for iteration 2 then iteration 1 or control, which means that my visualizations helped users understand what the code was doing, even if they had access to the code itself. As expected, I found that users had more consistent algorithm understanding if they had seen Julia before. Surprisingly, I found that algorithm understanding increased more for harder problems, which means that it was more useful in visualizing harder problems than it was for easier problems.

The most important result I found was in comparing the difference between good result runtime confidence and bad result runtime confidence. This is interesting because a large delta means that the user can easily see a difference between a good and bad algorithm. I found that this delta was larger in iteration 2 than in iteration 1, which means users found the second visualization most useful in differentiating between good and bad algorithms. Similarly, control had a much tighter spread of runtime confidence ratings, meaning they couldn't figure out whether they were doing well without the visualization. This shows that these visualizations are actually useful and that they help users.

Implementation:

Prior to testing, I investigated how easy it would be to get at the data in the first place. Luckily, it was very easy, and only consisted of modifications to one method in the Julia code. Below I describe what my investigation found (so if this visualization is made to work on real code in Julia, it can be done quickly) as well as how I implemented my visualization prototypes.

Julia:
To get the data (timestamp and data transferred) from each processor in Julia, I modified sync_msg to update a local array of data that is collated and parsed after the algorithm is run. Within sync_msg we have access to what is the remote reference that is. Afterwards, the data will be serialized to disk to a place where the visualizer can pick it up.

Visualization:
I prototyped my visualization using HTML5 and a vector art program named Artboard. Since the website uses D3, I think the actual implementation should use some combination of D3 and HTML5. My animations are static, but would be very easy to adapt to a dynamic animation based on the data from Julia.