CIS 115 - Introduction to Computing Science

Fall 2015

Scratch HPC Sum Project

DUE: Tuesday, October 20th by 10:00 PM via K-State Canvas

Instructions:

1. Follow the instructions below to build a new Scratch file to complete the assignment
2. You may complete this project in pairs of no more than two people if you so choose. However, each person in the group must submit a completed version of the project.
3. Submit a completed version of the assignment via the K-State Online gradebook. You will need to download a copy of your final project as a .sb2 file from Scratch and upload it to K-State Online. Make sure you include your teammate's name (if any) in your submission.

High Performance Computing

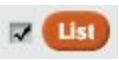
High Performance Computing is a very important part of our world. In this assignment, we're going to get first-hand experience working with HPC concepts and learning about some of the challenges involved in creating a program that works properly.

Starter File

To start, create a new project in Scratch 2.0 and select the Stage. We're going to use it as the basis for the first part of the project. Next, create the following variables (go to the Data palette and click the appropriate buttons):

- MultiSum
- Sum
- Time

We also need to create a list to store our data. So, on the Data palette, create a list variable called List as well.



Next, we need to set the initial state of our program. So, on the stage, arrange blocks so that, when the green flag is clicked, all the items in the list are deleted and all the variables are set to 0.

After that, we need to generate our data. To do so, we are going to insert 200 two digit numbers into the list, and calculate their sum. The easiest way to do that is to use a repeat (200) block, and add a random number at the end of the List. After we do that, we'll change the value of the Sum variable by the value that is the last item in the List.

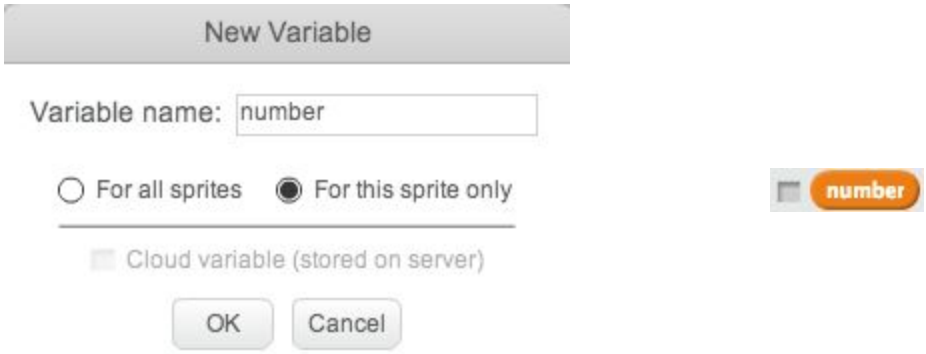
Finally, we need to set up a few other things. For this test, we want to be able to see how long each operation takes. To do that, we need to reset the Timer variable before we start, then save the value of the Timer in the Time variable once we are finished. The Timer variable is found on the light blue Sensing palette. Finally, between those two steps, we are going to broadcast and wait on a message. You can find messages on the Events palette. I named mine "start" but you may call yours whatever you want.

Lastly, make sure that you are able to see all your variables on the stage. You can do so by checkmarking the box next to their names in the Data palette. My stage looks like the screenshot below (I swapped the default cat sprite for a bat, just for a change of pace).

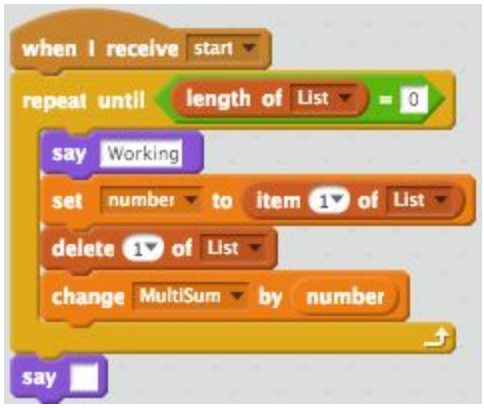
Worker Sprite

Next, click on the sprite itself. This is where we'll add the code that we need to perform our task. In this case, when it receives the "start" message, we want our sprite to add up all the items in the list and store that value in the MultiSum variable. Each time it adds a number to the total, it should remove it from the list.

Therefore, our sprite needs to have a variable created so it can keep track of which number it is storing. I called my variable number. Make sure you choose to make that variable for the sprite only, since we'll only want to use it there. Finally, since we don't really need to know the value it stores, you can uncheck it so it doesn't show up on the stage.

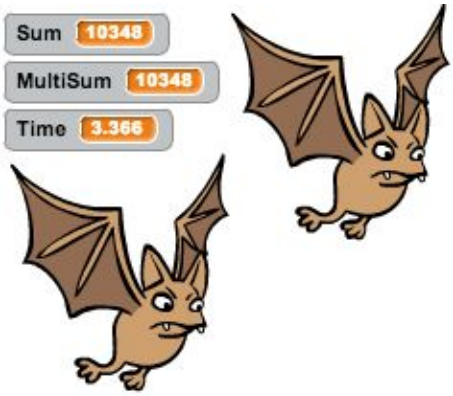


Now for the fun part. We can use blocks to get the first item from the list and store it in the number variable, then remove that item from the list. Once we've done that, we can add the value in the number variable to the MultiSum variable created earlier. To make this work, we need to continue doing these steps until the list is empty. Finally, while it is doing work, we want our sprite to say so, so we'll use a couple of say blocks to accomplish that. So, you'll end up with a block of code that looks like this:



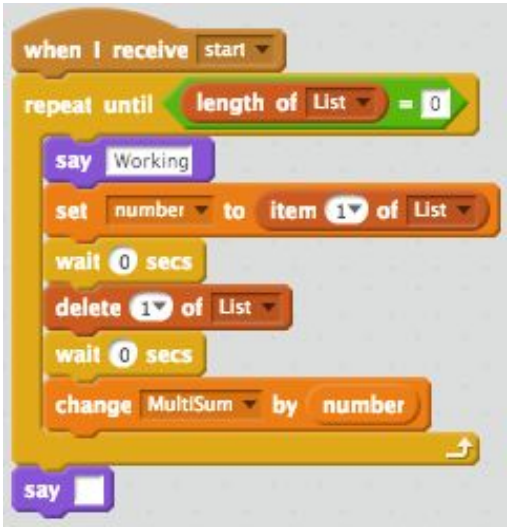
So, if you run the program, it should be able to correctly add up the numbers and tell you how long it took. Here's what it looks like when I do it:

Here's the fun part: instead of just using ONE sprite, why don't we use two? Let's make a copy of our sprite (right-click on it and select "duplicate") and run it again. Here's what I get:



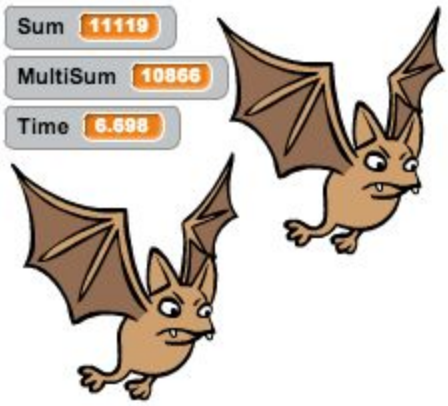
That works, and it even took half as long. So, we are now doing HPC just like in real life, right? (Well, sort of…)

HPC in Scratch



Scratch is a very beginner-friendly language, and behind the scenes it will try to account for and correct many common errors that arise when building large scale programs, including problems related to HPC. They do this by guaranteeing that each script cannot be interrupted once it starts running, except in a few situations. On a real computer, each instruction can be interrupted at any time, and you'll never know when it could happen. To simulate that in Scratch, we are going to use a couple of wait blocks between our actions. Now, our code should look something like this. Now let's run our program with just one sprite, using the code we have above:

It takes a bit longer, but it still works. And, if we duplicate our sprite, we can prove that it gives us the correct sum, just like it did before (or does it?):



Uh-oh! It ran faster, but we didn't get the correct answer. This is because the instructions get interrupted, and there's no guarantee that it will work properly when that happens. Here's what it looks like if we stack the instructions up in the order they might have happened:

Sprite 1	Sprite 2	List	MultiSum
Take the first number (number = 1)		[1, 2, 3, 4]	0
	Take the first number (number = 1)	[1, 2, 3, 4]	0
Remove the first number		[2, 3, 4]	0
	Remove the first number	[3, 4]	0
Add (number = 1) to MultiSum		[3, 4]	1
	Add (number = 1) to MultiSum	[3, 4]	2

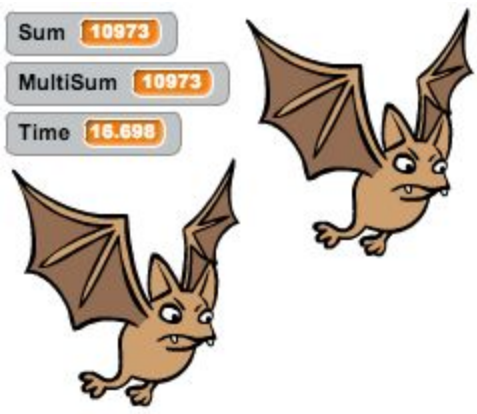
At this point, the MultiSum variable should be 3 instead of 2. This is because each sprite took the same number off the list first, then each sprite removed a different number from the list, accidentally removing 2 without it being included in the total.

Mutex

To fix this, we need to make sure that only 1 sprite can be reading and changing the list at a time. In HPC, we can use a structure called a “mutex” to accomplish this. Mutex is an abbreviated form of “mutual exclusion,” which is exactly what we want. Thankfully, a mutex is really easy to create in Scratch. All we need to do is add a variable to the stage called “Mutex” and make sure its value is equal to 0 initially.

Then, we use a simple if block to check and see what the value of the Mutex is. If it is 0, then no one is currently using the list and it is safe to do so. In that case, set the Mutex to 1, then access the list. Make sure you set the Mutex variable to 1 *immediately* after the if block or else it won’t quite work correctly. Once we are done, we set it back to 0. If the Mutex isn’t 0 when we check, then we just need to do nothing and check again in a bit.

Now we can run the program again with one and two sprites (be sure each sprite is the same):



That works, but using two sprites actually makes it take *longer* than before. When you run the program, you should hopefully notice that only one sprite is saying “working” at a time. Because of the Mutex, it prevents a sprite from being able to do anything while the other sprite is working. So, while this solves our problem correctly, it doesn’t actually make it run any faster.

Atomic Operations

To fix the problem, we need to reduce the amount of our program that is protected by the Mutex. We can do so by moving blocks around so that only the blocks that make up our *atomic operation* are between the two blocks that set and clear the Mutex.

An *atomic operation* is an operation that MUST NOT be interrupted before it is complete. You can find more information here:
http://en.wikipedia.org/wiki/Atomic_operation

Your Assignment

Your assignment is to modify the starter file we’ve created above to make the operation **run faster** when multiple threads are used compared to a single thread, while still **using a Mutex**. You will have to do so by reducing the amount of code that is contained within the atomic operation (between the two Mutex operations). If all goes well, it should run faster! In fact, it should even work with more than 2 threads. See how many threads you can use before it starts to get slower again!