18.337/6.338 Homework 1
Fall 2011

Due: Monday, September 26, 2011
You can email your work to the TA, or submit it in class, whichever you prefer.

In this assignment you will be exposed to different models of parallel computation. The goal is simply to say "hello world" through different tools and environments so you know how to access them.

We will use four combinations of tools and environments: MPI on a cluster, Star-P on a cluster, Julia on Amazon Elastic Compute Cloud (EC2), and Julia on multi-core. There is nothing too special about these combinations; it is also possible to run MPI on EC2 or Julia on a cluster, for example.

For each section of the assignment, copy a transcript of the interesting parts of your terminal session to a text file.

1 MPI on a cluster

Look at the class website:

`http://beowulf.csail.mit.edu/18.337/`

Read the information under the sidebar link "Darwin cluster", and make sure you can connect to `beagle.darwinproject.mit.edu` and `evolution.darwinproject.mit.edu`. Follow the instructions for "Setting up passwordless SSH".

Find the "MPI example" on the same page. Copy this text into an editor such as `emacs` or `vi` on `beagle` and save it with a file name ending in `.c`. Follow the instructions on the course site for compiling and running MPI programs. Try running the program several times. What do you notice?

On the homework page of the website, there is MPI code to evaluate an integral that equals π. There is code for the same program in both C and Fortran. Compile and run both versions.

2 Matlab with Star-P

Follow the "Star-P setup" instructions on the Darwin cluster page. We will use Star-P to solve the following problem. In fact we will tell you roughly how to solve it, and you will put the pieces together. If you wish, it is ok to do all of the work at the Matlab prompt and turn in a transcript of your session, instead of creating a `.m` file.

In the popular board game of Risk, it is common for an "attacker" to roll three dice and a "defender" to roll two. Let's call the sorted rolls of the attacker and defender $A_1 \geq A_2 \geq A_3$ and $D_1 \geq D_2$, respectively.

If $A_1 > D_1$ and $A_2 > D_2$ then the attacker "wins" 2.

If $A_1 \leq D_1$ and $A_2 > D_2$ then the attacker "wins" 1 and loses 1.

Also if $A_1 > D_1$ and $A_2 \leq D_2$ then the attacker "wins" 1 and loses 1.

If $A_1 \leq D_1$ and $A_2 \leq D_2$, then the attacker "loses" 2.

Run as many simulations as you wish. Figure out the probability of each of three events: A wins 2, A wins 1 and loses 1, A loses 2.

This can be done using Star-P's data-parallel functionality, demonstrating the distributed array approach to parallel computing. You can use something like Matlab's `ceil(6*rand(m,n))` to get dice rolls. The sorting step can be done with `sort(array, 2)`. Recall that in Matlab, comparisons apply to whole arrays elementwise, and the results of comparisons are ones and zeros that can be summed. Combining these features, simulation trials can be done with `sum(a(:,[1 2])>b, 2)`. A histogram is a nice way to obtain the final counts: `[count,y]=hist(sumofrolls, [0 1 2])`.

3 Julia on EC2

Follow all of the instructions under the "Amazon EC2" sidebar link on the course website. Find `peakflops()` on your instance.

4 Julia on multi-core

Read the "Julia example" on the Darwin cluster page. Also peruse the "Parallel Julia docs", especially the section on parallel loops (for now, only read the rest of it if you are interested).

The `beagle` and `evolution` machines themselves (as well as the compute nodes) have multiple cores, and so are each parallel computers on their own. Starting julia with `-p 4` will give you four processors on the local machine.

Port the C/MPI π integral program from section 1 to Julia. You can ignore the user input, printing, and timing parts, and just do the computation. You should write a function that looks like this:

```
function parallelpi(niter)
    h = 1.0/n
    YOUR CODE
end
```

The function can be written at the Julia prompt, and then called with `parallelpi(n)`. Or if you wish, you can write the function in a file and load it with `load("file.j")`.

Julia is experimental, and you may encounter problems. Just show us what you tried and what happened, and don't worry if you can't get it to work.

You might want to check whether the machine you're on is busy using `top`. If it is busy (i.e. there are some processes using close to 100% CPU), you can try connecting to a compute node with SSH instead.