MPI Lab 1: Hello, World!

The Message Passing Interface is a collection of tools that handle many common tasks from distributed computing. Unlike Hadoop, MPI is just a library of functions for moving data and spawning concurrent processes. Hence, you will find that MPI programming is much more engaging because it requires you to think not only about how to implement your software but also which distributed programming paradigm to use. With MPI you have nearly complete control over the design process of programming making MPI much more expandable.

I. Getting Started

In order to write your first program you will need to understand a few things about the way MPI works. Firstly, you will write and compile one program that calls the MPI library and this program will be launched on any desired nodes by the MPI interface. When your program first launches on all of the desired nodes, each instance of the application has exactly the same information, including command line arguments. Secondly, nodes can send and receive data with each other through communicators. When the program starts each node has a default communicator (MPI_COMM_WORLD) setup for it by MPI. Thirdly, the output from each instance of the program is aggregated at the terminal that you launch it from. Now on to semantics...

II. Writing the Hello Program

Now that you have a basic knowledge of MPI you can write your first program, Hello. First create a file, hello.c, this will contain the main function for your first program. Now in order to use MPI we need to import the MPI library, in this case we also want to call *printf* so we need the stdio library too. In hello.c add the following:

```c
#import <stdio.h>
#import <mpi.h>
```
Now that we have the MPI and IO functions we need we can begin writing our code. First let’s declare a variable to hold the rank of the instance.

```c
int main(int argc, char **argv) {

    // The current rank:
    int rank;

    But wait— How can we get our rank? Before we can use the default communicator, MPI_COMM_WORLD, it has to be initialized. We also need to initialize the command line arguments and setup the MPI environment. MPI provides the function `MPI_Init()` to do this for us. Once we’ve done that we can the use `MPI_Comm_rank()`, another MPI library function, to get the rank of the current instance at runtime.

    int main(int argc, char **argv) {
        ...

        // Initialize MPI:
        MPI_Init(&argc, &argv);

        // Get my Rank:
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    Now that all of the basic initialization steps are finished we can do something interesting. In this demo we will just have each instance print hello and goodbye to the screen with its process rank.

    int main(int argc, char **argv) {
        ...

        // Say Hello:
        printf("%i: Hello, World!\n", rank);

        // Say Goodbye:
        printf("%i: Goodbye!\n", rank);
```
That was pretty simple but we aren’t finished yet. MPI requires that we give it a chance to deallocate all of it’s resources and cleanup before we can exit. In order to tell MPI that we are finished and wont be calling anymore MPI functions, we simply call `MPI_Finalize()`. 

```c
int main(int argc, char **argv) {
    ...
    // Finish up:
    MPI_Finalize();
    // Return Success:
    return 0;
}
```

Now we’re done! Use `mpicc` to compile hello.c into an executable binary named Hello. You can watch the program run on a single machine by executing the Hello binary or you can run the program on a cluster with the command `mpiexec`.

```bash
mpicc -o Hello hello.c
mpiexec -n <number of processors> ./Hello
```

You may notice that the Hello and Goodbye messages are mixed together. Perhaps we want all of the machines to say hello before anyone is allowed to leave. MPI provides us with the function `MPI_Barrier()` which causes all nodes on a specified communicator to wait for each other before continuing. Make the following change, recompile your code and rerun Hello:

```c
int main(int argc, char **argv) {
    ...
    // Say Hello:
    printf("%i: Hello, World!\n", rank);
    // Wait for everyone to say hello:
    MPI_Barrier(MPI_COMM_WORLD);
}
```
Congratulations you just finished your first MPI program. Take a moment to try and figure out exactly what’s going on.

```c
#import <stdio.h>
#import <mpi.h>

int main(int argc, char **argv) {
    // The current rank:
    int rank;

    // Initialize MPI:
    MPI_Init(&argc, &argv);

    // Get my Rank:
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Say Hello:
    printf("%i: Hello, World!\n", rank);

    // Wait for everyone to say hello:
    MPI_Barrier(MPI_COMM_WORLD);

    // Say Goodbye:
    printf("%i: Goodbye!\n", rank);

    // Finish up:
    MPI_Finalize();

    // Return Success:
    return 0;
}
```