# Intro to MPI Exercise

In this workshop we will compile and run some very simple MPI programs in parallel on a single node. This will serve as an introduction to prepare you for running multi-node MPI jobs on your cluster later in the program.

Install some necessary applications before going forward:

```
$ sudo yum install -y openmpi-devel

$ . /etc/profile.d/modules.sh

$ module load mpi/openmpi-x86_64

$ which mpicc # should print /usr/lib64/openmpi/bin/mpicc

$ mpirun --version
```

Section 1: MPI Hello World

In this section, we'll parallelize the traditional C "hello world" program:

```
#include <stdio.h>
int main(int argc, char** argv)
{
        printf("hello world\n");
        return(0);
}
```

At a minimum, we'll need to add the MPI boilerplate to initialize MPI:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
        MPI_Init(&argc, &argv);
        printf("hello world\n");
        MPI_Finalize();
        return(0);
}
```

Put this code in a file (or grab a copy from #TODO) and compile it with the MPI wrapper:

```
$ mpicc mpi_hello.c -o mpi_hello
```

.. and run it in serial:

```
$ mpirun -np 1 ./mpi_hello
```

If all goes well, you should see “hello world” printed. Now let’s run in parallel:

```
$ mpirun -np 4 ./mpi_hello
```

You should see “hello world” printed once per process (4 in this case). You may see that some of the output is corrupted by two processes printing simultaneously. One way to solve this is to only print from a single node:

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv)
{
	int rank;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	if(rank == 0)
	{
		printf("hello world\n");
	}
	MPI_Finalize();
	return(0);
}
```

Recompile and rerun:

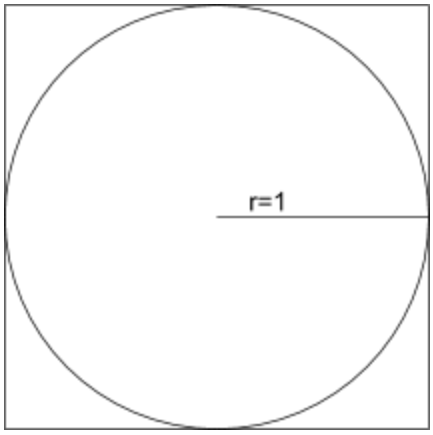
```
$ mpicc mpi_hello.c -o mpi_hello

$ mpirun -np 4 ./mpi_hello
```

This time, you should once again see only one message, despite running multiple processes.

**Section 2: Real World Example: Monte Carlo PI**

Now let’s apply MPI to a real problem: computing an approximate value for $\pi$. Since we have a cluster, we’ll use a Monte Carlo method which allows us to easily exploit the power of our cluster through randomness. The math for this is actually quite simple: Imagine a square dartboard with circle inscribed within it such that the diameter of the circle is the length of a side of the square (we’ll use a circle with a radius of 1):




We can observe that the ratio of the area of the circle to the area of the square is equal to some constant, $\pi$/4 (since the square's area is 2*2 = 4 and area_circle = $\pi$*r^2 = $\pi$). If we randomly place many points (darts) inside the square, we can count how many are also inside the circle (satisfy x^2+y^2 <= 1) vs the total number of points and compute an estimate for the value of $\pi$. To compute an accurate estimate for $\pi$ we'll want to use many points, so let's write a parallel program using MPI: