MPI

The purpose of MPI is to divide and manage computation among a group of processor nodes (e.g. cores or CPUs).

We'll cover the following

The Message Passing Interface (MPI) is a standard definition of core syntax and semantics of library routines that can be used to implement parallel programming in C (and in other languages as well). There are several implementations of MPI such as Open MPI, MPICH2 and LAM/MPI.

In the context of this tutorial, you can think of MPI, in terms of its complexity, scope and control, as sitting in between programming with Pthreads, and using a high-level API such as OpenMP.

The MPI interface allows you to manage allocation, communication, and synchronization of a set of processes that are mapped onto multiple nodes, where each node can be a core within a single CPU, or CPUs within a single machine, or even across multiple machines (as long as they are networked together).

One context where MPI shines in particular is the ability to easily take advantage not just of multiple cores on a single machine, but to run programs on clusters of several machines. Even if you don't have a dedicated cluster, you could still write a program using MPI that could run your program in parallel, across any collection of computers, as long as they are networked together. Just make sure to ask permission before you load up your lab-mate's computer's CPU(s) with your computational tasks!

Here is a basic MPI program that simply writes a message to the screen indicating which node is running.

```
// mpicc go_mpi.c -o go_mpi
// mpirun -n 4 go_mpi
```

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

int main(int argc, char *argv[])
{
  int myrank, nprocs;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

  printf("I am node %d of %d\n", myrank, nprocs);

  MPI_Finalize();
  return 0;
}
```

```
plg@wildebeest:~/Desktop$ mpicc go_mpi.c -o go_mpi
plg@wildebeest:~/Desktop$ mpirun -n 4 go_mpi
I am node 0 of 4
I am node 2 of 4
I am node 1 of 4
I am node 3 of 4
```

The basic design pattern of an MPI program is that the **same code** is sent to all nodes for execution. It's by using the `MPI_Comm_rank()` function that you can determine which node is running, and (if needed) act differently. The `MPI_Comm_size()` function will tell you how many nodes there are in total.

MPI programs need to be compiled using `mpicc`, and need to be run using `mpirun` with a flag indicating the number of processors to spawn (4, in the above example).

MPI$_{\text{Reduce}}$

We saw with OpenMP that we can use a **reduce** directive to sum values across all threads. A similar function exists in MPI called `MPI_Reduce()`.

An Example: Estimating pi using dartboard algorithm

```c
// Estimating pi using the dartboard algorithm
// All processes contribute to the calculation, with the
// master process averaging the values for pi.
// We then use mpc_reduce to collect the results
//
// mpicc -o go mpi_pi_reduce.c
// mpirun -n 8 go

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

```
#define MASTER 0         // task ID of master task
#define NDARTS 1000      // # dart throws per round
#define NROUNDS 10     // # of rounds of dart throwing

// our function for throwing darts and estimating pi
double dartboard(int ndarts)
{
  double x, y, r, pi;
  int n, hits;
  hits = 0;

  // throw darts
  for (n = 1; n <= ndarts; n++)  {
    // (x,y) are random between -1 and 1
    r = (double)random()/RAND_MAX;
    x = (2.0 * r) - 1.0;
    r = (double)random()/RAND_MAX;
    y = (2.0 * r) - 1.0;
    // if our random dart landed inside the unit circle, increment the score
    if (((x*x) + (y*y)) <= 1.0) {
      hits++;
    }
  }

  // estimate pi
  pi = 4.0 * (double)hits / (double)ndarts;
  return(pi);
}

// the main program
int main (int argc, char *argv[])
{
  double my_pi, pi_sum, pi_est, mean_pi, err;
  int task_id, n_tasks, rc, i;
  MPI_Status status;

  // Obtain number of tasks and task ID
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&n_tasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&task_id);
  //  printf ("task %d of %d reporting for duty...\n", task_id, n_tasks);

  // different seed for random number generator for each task
  srandom (task_id);

  mean_pi = 0.0;
  for (i=0; i<NROUNDS; i++) {
    // all tasks will execute dartboard() to calculate their own estimate of pi
    my_pi = dartboard(NDARTS);

    // now we use MPI_Reduce() to sum values of my_pi across all tasks
    // the master process (id=MASTER) will store the accumulated value
    // in pi_sum. We tell MPI_Reduce() to sum by passing it
    // the MPI_SUM value (define in mpi.h)
    rc = MPI_Reduce(&my_pi, &pi_sum, 1, MPI_DOUBLE, MPI_SUM,
            MASTER, MPI_COMM_WORLD);

    // now, IF WE ARE THE MASTER process, we will compute the mean
    if (task_id == MASTER) {
      pi_est = pi_sum / n_tasks;
      mean_pi = ( (mean_pi * i) + pi_est ) / (i + 1); // running average
```

```
      err = mean_pi - 3.14159265358979323846;
      printf("%d throws: mean_pi %.12f: error %.12f\n",
         (NDARTS * (i + 1)), mean_pi, err);
    }
  }
  if (task_id == MASTER)
    printf ("PS, the real value of pi is about 3.14159265358979323846\n");

  MPI_Finalize();
  return 0;
}
```

Here we run it with just one parallel process:

```
plg@wildebeest:~/Desktop/mpi$ time mpirun -n 1 go
1000 throws: mean_pi 3.088000000000: error -0.053592653590
2000 throws: mean_pi 3.104000000000: error -0.037592653590
3000 throws: mean_pi 3.101333333333: error -0.040259320256
4000 throws: mean_pi 3.120000000000: error -0.021592653590
5000 throws: mean_pi 3.124800000000: error -0.016792653590
6000 throws: mean_pi 3.127333333333: error -0.014259320256
7000 throws: mean_pi 3.134285714286: error -0.007306939304
8000 throws: mean_pi 3.128500000000: error -0.013092653590
9000 throws: mean_pi 3.132444444444: error -0.009148209145
10000 throws: mean_pi 3.119600000000: error -0.021992653590
PS, the real value of pi is about 3.14159265358979323846

real    0m0.032s
user    0m0.020s
sys 0m0.012s
```

Now let's run it with 4:

```
plg@wildebeest:~/Desktop/mpi$ time mpirun -n 4 go
1000 throws: mean_pi 3.105000000000: error -0.036592653590
2000 throws: mean_pi 3.122500000000: error -0.019092653590
3000 throws: mean_pi 3.122000000000: error -0.019592653590
4000 throws: mean_pi 3.137750000000: error -0.003842653590
5000 throws: mean_pi 3.143600000000: error 0.002007346410
6000 throws: mean_pi 3.140166666667: error -0.001425986923
7000 throws: mean_pi 3.142000000000: error 0.000407346410
8000 throws: mean_pi 3.140250000000: error -0.001342653590
9000 throws: mean_pi 3.136666666667: error -0.004925986923
10000 throws: mean_pi 3.135000000000: error -0.006592653590
PS, the real value of pi is about 3.14159265358979323846

real    0m0.034s
user    0m0.044s
sys 0m0.024s
```

We see the final error is much reduced. Each of the 4 processes (which are parallelized across the cores of my CPU) contributes an estimate of pi, which are then averaged by the master process to come up with the final estimate of pi.

For more on parallel programming, visit the links provided in the next lesson. Some interesting exercises await you as well. Good luck!