

Topic 1: Introduction (continued)

In this lecture we continue discussion of MPI using the homework problem as an example.

Monte Carlo calculation of π

In the last lecture the value of π was represented or “modeled” by the definite integral

$$\pi = \int_0^1 dx \frac{4}{1+x^2},$$

with the simplifying approximation

$$\int_0^1 dx f(x) \simeq h \sum_{i=1}^n f(h(i-0.5)), \quad h \equiv \frac{1}{n}.$$

A different representation or “model” for π is

$$\pi = \frac{4 \times \pi R^2}{(2R)^2} = \frac{4 \times \text{Area of a circle}}{\text{Area of enclosing square}}.$$

As a simplifying approximation we can use a random or *Monte Carlo* method: Choose points randomly inside of the square. The points should be *uniformly distributed*, that is, each location inside the square should occur with the same probability. Then the simplifying approximation which allows us to compute π is

$$\frac{4 \times \text{Area of a circle}}{\text{Area of enclosing square}} \simeq \frac{4 \times \text{Number of points inside the circle}}{\text{Total number of points}}.$$

A simple algorithm to implement this Monte Carlo calculation is:

- Choose the radius of the circle $R = 1$, so the enclosing square has side $2R = 2$ and area $(2R)^2 = 4$.
- Generate a random point inside the square.
 - Use a random number generator to generate a *uniform deviate*, that is, a real number r in the range $0 < r < 1$. The x coordinate of the random point is $x = 2r - 1$.
 - Repeat to find the y coordinate.
 - If $x^2 + y^2 < 1$ then the point is inside the circle of radius 1, so increment the number of inside points.
 - Increment the total number of points.
- Repeat until the desired number of points have been generated.
- Compute

$$\pi = \frac{4 \times \text{Number of points inside the circle}}{\text{Total number of points}}.$$

Here is the C/C++ implementation of this algorithm:

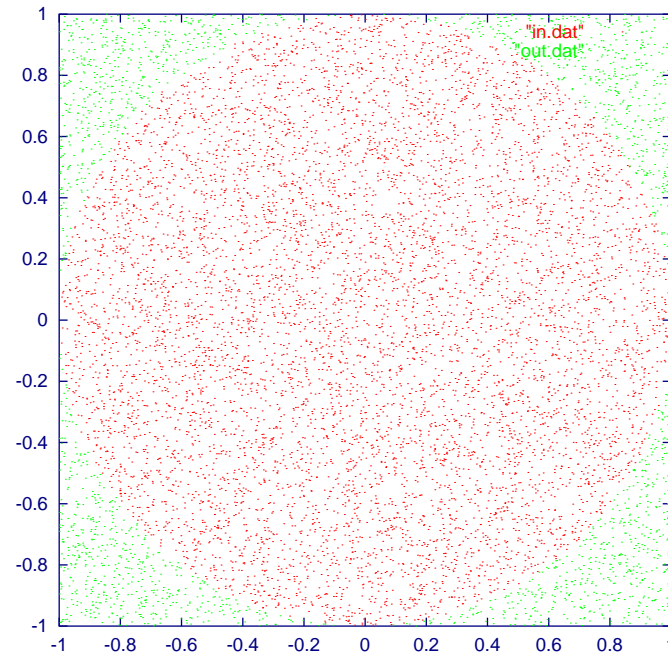
```
#include <stdlib.h>

int in, out, i, iter, done;
double x, y, pi, error, epsilon;

printf("Enter desired accuracy: ");
scanf("%1f", &epsilon);

done = in = out = 0;
iter = 0;
while (!done) {
    iter++;
    x = ((double) rand() / RAND_MAX) * 2 - 1;
    y = ((double) rand() / RAND_MAX) * 2 - 1;
    if (x*x + y*y < 1.0)
        in++;
    else
        out++;
    pi = (4.0 * in) / (in + out);
    error = fabs(pi - 3.141592653589793238462643);
    done = (error < epsilon || (in + out) > 1000000);
}
printf("pi = %23.30f, No. of points = %d\n", pi, iter);
```

The figure plots data from a Monte Carlo run with 10,000 points.



MPI version of program

The algorithm used in the MPI version of the program is:

- The user specifies `numprocs` processes (CPU's). These are divided into two groups:
 - The `worker_group` contains processes with ranks `0,1,...,numprocs-2`. Process 0 reads input and prints output.

- The last process with rank `server=numprocs-1` generates “chunks” of `CHUNKSIZE` random integers when asked by any worker.
- The server operates in a loop which
 - signals it is waiting for a request,
 - when it receives a `request=1` generates `CHUNKSIZE` random integers and sends them to the worker which requested them, or
 - quits the loop if *any* worker signals that no more integers are needed (`request=0`).
- Each worker initially sends a request to the server, and then enters a loop in which it
 - waits until it receives a chunk of random integers from the server,
 - generates `CHUNKSIZE/2` random points and totals the number of points inside and outside the circle of radius 1, just like the serial program, and
 - checks whether sufficient points have been generated and signals the server appropriately.

Compared with the rectangle rule program, this Monte Carlo π program calls the following additional MPI functions:

- to create the worker group

```
MPI_Comm world, workers;
MPI_Group world_group, worker_group;
int ranks[1];
server = numprocs-1;
world = MPI_COMM_WORLD;
MPI_Comm_group( world, &world_group );
ranks[0] = server;
MPI_Group_excl( world_group, 1, ranks, &worker_group );
MPI_Comm_create( world, worker_group, &workers );
MPI_Group_free(&worker_group);
```

Note the two types of MPI objects involved here:

- *Groups* of type MPI_Group, which represent groups of processes, and
- *Communicators* of type MPI_Comm, which represent a *context* and the group with which it is associated. A context is essentially a label which identifies messages which groups members exchange with one another.

The `workers` communicator is constructed by

- accessing the `world_group`,
- storing the ranks of processes *not* in the worker group in the array `ranks`,
- creating the `worker_group` from the `world_group` by excluding the server,
- creating a communicator for the workers, and

- freeing the `worker_group`. (This is optional and does not actually destroy the `worker_group` until all associated communicators are also freed—so you could omit the `MPI_Group_free` function call.)

- A process can send a message by calling

```
int MPI_Send( void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm )
```

e.g., `MPI_Send(&request, 1, MPI_INT, server, REQUEST, world)` sends `request` which consists of 1 integer to the `server` with message id or *tag* = `REQUEST`. The sender and receiver are identified by the communicator `world`. The return value of `MPI_Send` is an error code.

- A process can signal that it is waiting for a message by calling

```
int MPI_Recv( void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm, MPI_Status *status )
```

There is one additional argument of type `MPI_Status` compared with `MPI_Send`: when the message is actually received this output argument is filled with information on the message, including the source, tag, and length.

- `MPI_Allreduce` combines values from all processes in a group and distribute the result back to all processes. It is basically equivalent to calling `MPI_Rreduce` followed by `MPI_Bcast`.