Assignment C/MPI: Back to Basis with π

S. Varrette, V.Plugaru and P. Bouvry
<Firstname.Lastname@uni.lu>

Version 1.0

The following information may interest you:

- *http://www.idris.fr/data/cours/parallel/mpi/mpi1_aide_memoire_C.html*
- *a trapeze of big length a, short length b and height h has area* $h \times \frac{a+b}{2}$.

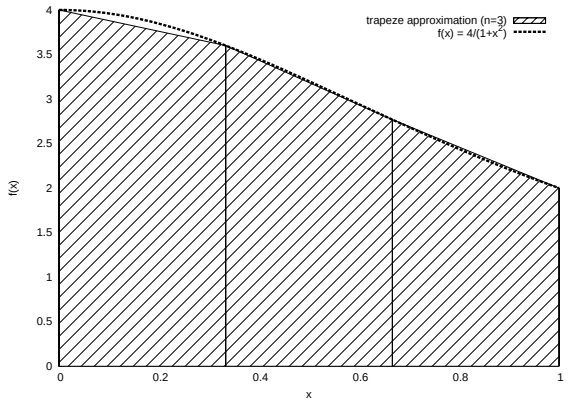
Exercise 1 *Pi computation using trapezium rule*

The purpose of this exercise is to compute the value of π using an easy and customized method relying on an integral approximation. It introduces two MPI primitives: `MPI_Bcast` and `MPI_Reduce`.

More precisely, let $f(x) = \dfrac{4}{1+x^2}$. Then:

$$\int_0^1 f(x)dx = 4\int_0^1 \frac{1}{1+x^2}dx = 4\left[\arctan(x)\right]_0^1 = \pi$$

We will approximate this integral using the trapezium rule as illustrated in the following figure:



Assuming the interval [0,1] to be divided into n subintervals of equal size $h = \frac{1}{n}$, then

$$\int_0^1 f(x)dx \simeq \frac{1}{2n}\sum_{i=0}^{n-1}\left(f\left(\frac{i}{n}\right) + f\left(\frac{i+1}{n}\right)\right)$$

1. Find a upper bound for the error made by the approximation.
   Hint: find the real constant M such that $\forall x \in [0,1], \|f''(x)\| \leq M$. The error made is then $\leq M\frac{1}{12n^2}$.
2. Implement a sequential program in C that asks the user for the value of n and compute the value of π using the trapezium rule with n subintervals together with the error induced by the approximation.

3. Analyze a parallel version of this program using Foster's methodology and implement your solution in MPI.
4. Benchmark your program for various number of processors and plot your data using `gnuplot`.
   Hint: you may use a barrier synchronization using `MPI_Barrier` at the beginning of your program before using `MPI_Wtime` and/or `MPI_Wtick`.

Exercise 2 *Extension with Simpson's Rules*

Simpon's rule is a better numerical integration algorithm than the trapezium rule. Assuming again that the interval [0,1] is divided into $n = 2m$ subintervals of equal size $h = \frac{1}{n} = \frac{1}{2m}$, then:

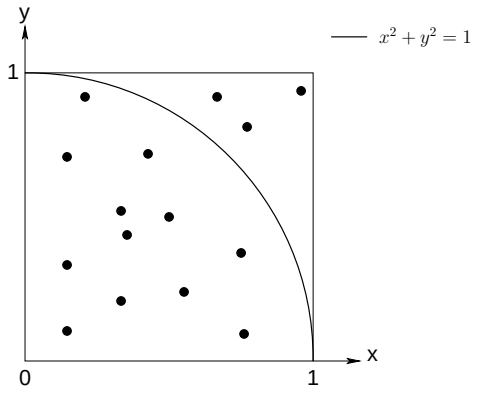
$$\int_0^1 f(x)dx \simeq \frac{1}{3n}\left[f(0) + f(1) + 2\sum_{i=1}^{m-1} f\left(\frac{i}{m}\right) + 4\sum_{i=1}^{m} f\left(\frac{2i-1}{2m}\right)\right]$$

1. Implement a sequential program in C that asks the user for the value of m and compute the value of π using simpson's rule with $n = 2m$ subintervals together with the error induced by the approximation.
2. Analyze a parallel version of this program using Foster's methodology and implement your solution in MPI.
3. Benchmark your program for various number of processors and plot your data using `gnuplot`.

Exercise 3 *Evaluation by a Monte-Carlo method*

Another approach for evaluating π is to rely on a Monte-Carlo method. Such a method solves a problem through the use of statistical sampling.

Here we will use geometrical analogy. A quarter circle of radius r has area $\frac{\pi}{4}r^2$ while a square with sides of length r has areas r^2. The ratio between those two area is therefore $\frac{\pi}{4}$. We can use random numbers to perform a similar estimation. The next figure illustrates the approach with $r = 1$. The idea is to generate n random points (x, y) uniformly distributed inside the square (*i.e* such that $0 \leq x, y \leq 1$).



Then we keep track of the points that fall inside the quarter circle *i.e* for which $x^2 + y^2 \leq 1$. It follows that:

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

For instance on the preceding figure, 12 of 15 points randomly chosen from the unit square are inside the circle, resulting in an estimate of $3,2$ for π.

Given estimated value e and correct value c, the absolute error induced by this method is $\frac{|e-a|}{a}$. The function $\frac{1}{2\sqrt{n}}$ closely approximates the absolute error of this Monte-Carlo method. Note that this approach can be theoretically proved using the mean value theorem which states that:

$$\int_a^b f(x)dx = (b-a)\bar{f}$$

1. Implement a sequential program in C that asks the user for a desired accuracy ϵ ($\epsilon = 0,001$ for instance) and computes the value of π using this Monte-Carlo method with an absolute error bounded by ϵ (or such that `MAX_NB_RANDOM_POINT` points have been generated). Your program should better rely on `random` instead of `rand` to generate random numbers between 0 and `RAND_MAX`. It would even be better to use the `rand48` family available on BSD-like machines.
2. Plot a graphical view of the two classes of points (under the quarter circle or not) using `gnuplot`.

Monte-Carlo algorithm often migrates easily onto paralel systems as they have a negligible amount of interprocessor communications. When this is the case, p processors can be used either to find an estimate about p times faster *or* to reduce the error of the estimate by a factor of $\sqrt{p}$.

One challenge is then the development of parallel random number generator that ensure (among other constraints) uniformly distributed sequences with no correlation between them. Here we propose to transform a sequential pseudo-random number generator (PRNG) into a parallel one using a **Manager-Worker** method: a single process (the *manager*) is responsible for generating a sequence of random numbers upon request and send them to the workers.

If this approach presents several drawbacks (most important about scalability), it will help to introduce the MPI primitives to create *groups* (of type `MPI_group`), new communicators (of type `MPI_comm`) and to perform a global reduction. The behaviour you'll have to implement is the following:

- Let p be the number of processes involved in the MPI execution. They are divided into two groups:
  - the `worker_group` that contains processes with ranks $0, 1, \cdots, p-2$. Process 0 reads input and prints output. In particular, he will broadcast the value of ϵ to the other processes.
  - The last process with rank `server`=$p-1$ 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 sequential program, and

– checks whether sufficient points have been generated and signals the server appropriately.

3. Construct a `worker` communicator by:

   - accessing the world group `MPI_COMM_WORLD`
   - storing the ranks of processes not in the worker group in the array ranks (see `MPI_Comm_group`)
   - creating the worker group from the world group by excluding the server (see `MPI_Group_excl`)
   - creating a communicator for the workers (see `MPI_Comm_create`)

   Eventually, you may free the `worker_group` using `MPI_Group_free` (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.)
4. Implement a parallel version of the Monte-Carlo method for estimating π using the protocol presented above. You'll have to use the functions `MPI_Send` and `MPI_Recv` seen during the lecture together with the function `MPI_Allreduce` (which combines values from all processes in a group and distribute the result back to all processes. It is basically equivalent to calling `MPI_Reduce` followed by an `MPI_Bcast`.
5. Benchmark your program for various number of processors and plot your data using `gnuplot`.