# Simple MPI parallelism

Dr. Christopher Marcotte — *Durham University*

In this exercise we're going to compute an approximation to the value of $\pi$ using a simple Monte Carlo method. We do this by noticing that if we randomly throw darts at a square, the fraction of the time they will fall within the inscribed circle approaches $\pi$. The following argument should be familiar to you; recall when we did the same calculation using OpenMP.

Consider a square with side-length $2r$ and an inscribed circle with radius $r$, as in Figure 1.

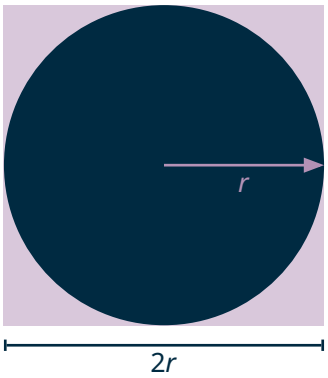


Figure 1: Square with inset circle of radius $r$.

The ratio of areas is $A_{\text{circle}}/A_{\text{square}} = (\pi r^2)/(4r^2) = \pi/4$. If we therefore draw $X$ uniformly at random from the distribution $U(0, r) \times U(0, r)$, then the probability that $X$ is in the circle is given by

$$p_{\text{in}} = (4r^2)^{-1} \int_{-r}^{+r} \int_{-r}^{+r} \Theta(r^2 - x^2 + y^2)\,\mathrm{d}x \times \mathrm{d}y.$$

We can therefore approximate $\pi$ by picking $N$ random points $X \sim U(0, r) \times U(0, r)$ and counting the number that fall within the circle $N_{\text{in}}$,

$$\tilde{\pi} = 4\left(\frac{N_{\text{in}}}{N}\right) \approx \pi.$$

Similarly, this implies that the error of the approximation for $N$ samples is $|\tilde{\pi} - \pi| = |4(N_{\text{in}}/N) - \pi|$.

## Serial implementation

This exercise is all about adapting our prior parallel approach with OpenMP and threads for MPI and processes.

On Hamilton, you will need to load the relevant compiler module and then build the executable with `make`. The executable can be run with `./calc_pi N` where `N` is the total number of random points sampled to approximate $\tilde{\pi}$.

### Convergence

Of preeminent importance for Monte Carlo methods is the convergence of the numerical approximation to the true value. Many careers have been made many times over by substantial contributions to methods which improve either the computational efficiency of these approaches or their analytical bounds. In this exercise we humbly verify the convergence and timing of the serial implementation, as a baseline for our future MPI changes.

***Hint***

You can execute a sequence of commands in your batch script (just make sure to request enough time for them all), rather than running each one in its own job. In fact, a batch script is just a bash script with additional header information used by Slurm – you can commit all manner of abuses in that notoriously ill-considered language.

***Exercise***

Compile and run the serial code for different choices of $N$ and plot the error as a function of $N$. What relationship do you observe between the accuracy of the approximate result and $N$? Verify that the runtime scales quasi-linearly with $N$ – you needn't do this rigorously, just use `time ./serial N`. If you seek to reach a specified accuracy for the approximation, i.e. $\varepsilon : |\tilde{\pi} - \pi)| < \varepsilon$, how many samples do you need (as a function of $N$)?

***Challenge***

Write an on-line estimate for $\tilde{\pi}$ and the variance of the estimate, such that for sample $x_k$, $t_k = (\|x_k\| < 1) \in \{0, 1\}$, we generate $\mu_k$ and $\sigma_k$ according to:

$$\mu_{k+1} = \frac{k}{k+1}\mu_k + \frac{1}{k+1}t_k, \quad S_{k+1} = S_k + \frac{k}{k+1}(t_{k+1} - \mu_k)^2.$$

By printing $\mu_k, \sigma_k = \sqrt{\frac{S_k}{k}}$ at each iteration (or every few iterations) we retrieve all the samples from a single progran run.

## Parallelisation with MPI

As we recall from the OpenMP exercise – this calculation is very amenable to parallelism. Adapting our approach from shared memory parallelism with OpenMP to distributed memory parallelism with MPI requires a bit of work.

***Hint***

Remember that the first thing every MPI program should do is to *initialise* MPI with `MPI_Init(...)` and the final thing every MPI program should do is *finalise* MPI with `MPI_Finalize(...)`.

***Exercise***

Add MPI initialisation and finalisation to the `main(...)` function of the serial code.

Determine the size of `MPI_COMM_WORLD` as well as the rank of the current process, and print these values out on every process.

Now compile and then run the executable with two processes using `mpirun`. What do you observe? Does the program behave as you expect?

So far all we've done is run a completely serial program on multiple machines. This is not very useful, except to waste energy. There are three fundamental parts to this program:

1. the sample generation,
2. the mapping from a sample to whether it is in the circle,
3. the summation of the in-circle samples.

Each presents some unique challenges for distributed memory parallelism with MPI compared to shared memory parallelism with OpenMP.

### Parallelising the random number generation

You may recall that we struggled to parallelise the random number generation in OpenMP because such methods are *stateful* and that hidden state is modified on every subsequent call to `drand48()`. In the previous exercise, we

resorted to our own random number generation which was OpenMP thread-safe, but not especially *random*. In MPI, since there is no shared memory, the situation is slightly different.

***Exercise***

The first thing to do is to ensure that the different processes use *different* random numbers. These are generated using the C standard library's pseudo-random number generator. The initial state is *seeded* in the function `calculate_pi(...)`.

Modify the code so that the seed depends on which process is generating the random numbers. Run again on two processes, do you now see that the results are different depending on the process?

***Hint***

The `rank` of a process is a unique identifier.

***Warning***

The approach we use here does not produce statistically uncorrelated random number streams. This does not really matter for this exercise, it just means that the *effective* number of Monte Carlo samples is lower than the $N$ we specify.

If you need truly independent random number streams, then the different approaches described here give more information on how to achieve it. It is a very deep topic, but frequently more interesting to cryptographic than scientific or high-performance computing.

## Dividing the work

Now we have different processes using differents seeds we need to divide up the work such that we take `N` samples in total (rather than `N` samples on each process).

***Exercise***

Modify the `calculate_pi` function such that the samples are (reasonably) evenly divided between all the processes. After this you're producing a partial result on every process.

Finally, combine these partial results to produce the full result by summing the number of points found to be in the circle across all processes.

***Hint***

Remember that you wrote a function in the ring reduction exercise to add up partial values from all the processes. Alternately, you may find the function `MPI_Allreduce` useful.

***Exercise***

Test your code running with $p = 1, 2, 4, 8, 16$ processes (and one process per core!). Produce a plot of the runtime as a function of the number of cores. What observations can you make? Is the scaling and efficiency what you expect?

***Hint***

It is good practice when using MPI that every function which is *collective* explicitly receives as an argument the communicator. If you find yourself explicitly referring to one of the default communicators (e.g. `MPI_COMM_WORLD`) in a function, think about how to redesign the code so that you pass the communicator as an

argument, e.g. `MPI_Comm comm`. This way, if your code is changed to use different communicators, it will work transparently.

***Exercise***

Perform a *weak* scaling analysis of the Monte Carlo code (i.e., make the number of samples scale with the number of processes $N(p) = N_1 p$, where $N_1$ is chosen to tax each process but not overwhelm it). Compute the speedup $S(p) = t(1)/t(p)$, and fit Gustafson's Law to it to determine the serial fraction $f$. What serial fraction do you find? Plot your weak scaling efficiency $E(p)$, and compare to the weak scaling efficiency from Gustafson's Law with your fit $f$.

## Quasi Monte Carlo

You may question whether completely random sampling is 'optimal' in some sense – e.g. the extremal power of the asymptotic convergence rate – and seek to implement an alternative sampling procedure for comparison to Monte Carlo convergence. One of the simplest methods is the so-called *Golden Sample* which was popularized (though I am not sure if it was discovered by) Martin Roberts in his blog entitled *The Unreasonable Effectiveness of Quasirandom Sequences*.

The idea is to use a sufficiently irrational number to sample a space deterministically, at low cost, with only some modular arithmetic involved[1]. This is largely the domain of quasi-Monte Carlo approaches, where a low-discrepancy sequence is used to sample a function so that the convergence is $O(N^{-d})$ where $d > 1/2$, as in Monte-Carlo.

To implement this sampling protocol, you would write a function which can generate the basis $\alpha = (\varphi_d^{-1}, \varphi_d^{-2}, \ldots)$ up to dimension $d$, where $\varphi_d$ is the unique positive root of the polynomial

$$x^{d+1} = x + 1, \tag{1}$$

e.g., where $x > 0$. For $d = 1$ then $\varphi_1 = \frac{\sqrt{5}+1}{2}$ (the golden ratio), and for $d = 2$ then $\varphi_2 = \sqrt[3]{\frac{9+\sqrt{69}}{18}} + \sqrt[3]{\frac{9-\sqrt{69}}{18}}$ (the plastic number), and so on and so forth. You can compute these values for any $d$ by applying the bisection procedure to Equation 1 over the interval $x \in (0, \varphi_{d-1})$ – as $d$ increases, the value of $\varphi_d$ decreases.

From the basis $\alpha$, you generate the $n$th sample $t_n = (n \cdot \alpha) \bmod 1$, and run your sampling convergence again. Note that this sampling is *not* sequential – $t_n$ is independent of all the previous samples, and all the future samples – so you can trivially parallelize the sample generation across $p$ processes using $t'_n = ((np + p') \cdot \alpha) \bmod 1$, where $t'_n$ is the $n$th sample on process $p' \in [0, p)$.

***Challenge***

Implement the Golden sampling protocol and verify that you produce an approximation of $\pi$ – what convergence do you observe? What convergence do you expect? Time your implementation with a fixed $N$ and compare it to the random sampling Monte Carlo you did earlier. Determine if the generation of $N$ golden samples is faster or slower than generating $N$ purely random points. Perform a *weak* scaling analysis with the Golden sampling quasi-Monte Carlo code (i.e., make the number of samples scale with the number of processes $N(p) = N(1)p$, $N(1)$ is chosen to tax each process but not overwhelm it). Compute the speedup $S(p) = t(1)/t(p)$, and fit Gustafson's Law to it to determine the serial fraction $f$. What serial fraction do you find? Plot your weak scaling efficiency against $p$, and compare to weak scaling efficiency from Gustafson's Law.

[1] And the computation of a scalar root for dimensions higher than 3.

***Aims***

- Consideration of random number generation in MPI parallelization contexts
- Comparison of computation of $\pi$ using MPI to OpenMP in previous exercise
- Explicit consideration of communication cost in an otherwise embarrassingly parallel problem