Parallel and High Performance Computing

Dr. Pablo Antolín

Series 4

March 20 2025

MPI

Different parallelizations of the π reduction

The code is in the `math454-phpc/exercises-2025` repository on `gitlab`. Just pull the latest version and you will get the `lecture_04/pi` folder.
The API documentation for MPI can be found in https://rookiehpc.github.io/mpi/docs/.

1 First steps with MPI

Exercise 1.1: ***MPI: Hello, World!***

- Initialize/finalize properly MPI.
- To compile you will have to adapt the `Makefile` to use `mpicxx` (or `mpiicpc`).
- Print out the number of processes and the rank of each process.
- Write a batch script to run your parallel code:

```
#!/bin/bash
#SBATCH --qos=math-454
#SBATCH --account=math-454
#SBATCH -n <ntasks>
module purge
module load <compiler> <mpi library>
srun <my_mpi_executable>
```

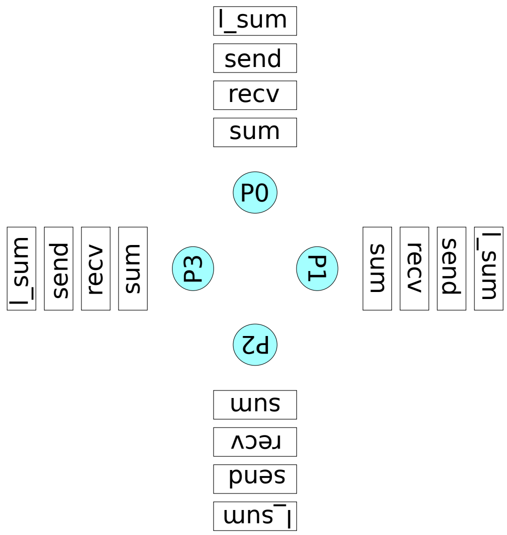
Note: *To use MPI on the cluster you first have to load a MPI implementation through the module openmpi (`module load gcc openmpi`) or intel-oneapi-mpi (`module load intel intel-oneapi-mpi`). In addition in the SLURM environment you should use `srun` instead of `mpiexec` or `mpirun`.*

2 Using a Ring to communicate

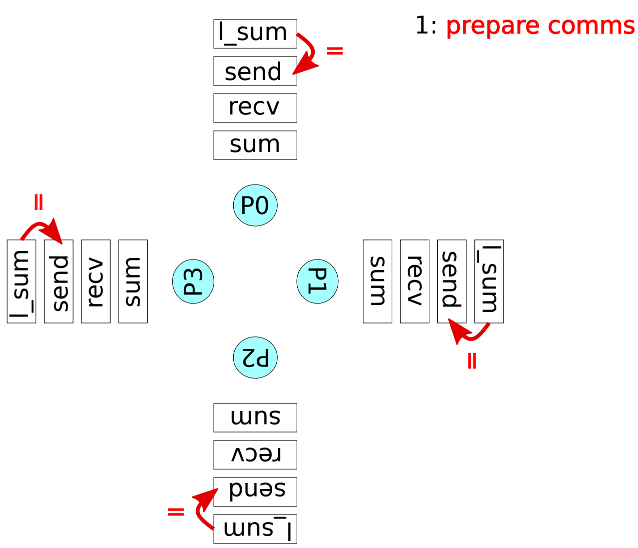
In these exercises, every process will compute a portion of the π integral. And then the partial sum will be moved around the ring in order for every process to be able to compute the full integral by summing all the partial sums. To explain how the point to point communications work on a ring, we will use an example with 4 processes:

1 Each process computes its local partial sum and stores it in `l_sum`.

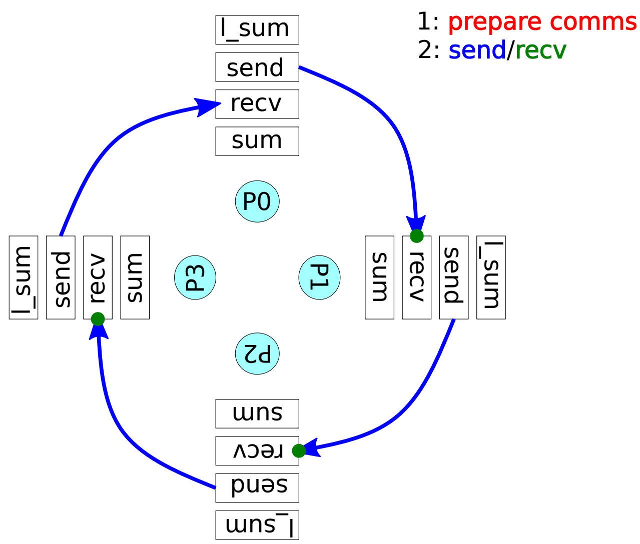
2 We start with each processes having a local partial sum in `l_sum`, a `send` and `recv` buffers, and finally a place to store the final total `sum`. `sum` should be initialized to the value of `l_sum`.



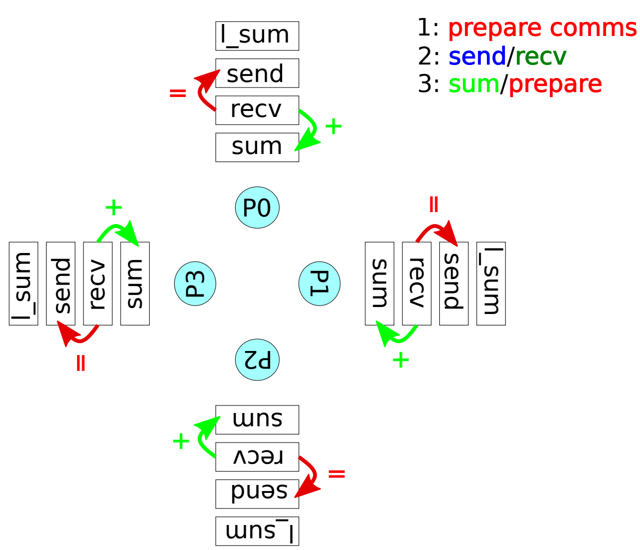
3 Each process takes its `l_sum` and copies it in the `send` buffer.



4 Then, each of them sends the data to the next process and receives data from the previous one. ***Note:*** *in a loop the next process' rank is "*`(prank + 1) % psize`*" and the previous one is "*`(prank - 1 + psize) % psize`*", % being the modulo operator in C/C++,* `prank` *the local rank of each process, and* `psize` *the total number of processes.*



5. Once the data are received, the data from `recv` can be accumulated in `sum` and copied to `send`, overwriting the previous value in the buffer, in order to be sent to the next process.



6. Now we repeat steps 4 and 5, enough times (`psize - 2` more, to be precise) for each process to see each local sum pass though. In total the steps 4 and 5 are executed `psize - 1` times.

Exercise 2.1: π ***MPI Ring (point to point synchronous)***

- Split the integral calculation among the processes.
- Implement a ring to communicate the partial sum among the processes using `MPI_Ssend` and `MPI_Recv`.

  Remember: *each MPI process runs the same code!*

Exercise 2.2: π ***MPI Ring (point to point synchronous sendrecv)***

Modify the previous exercise to use `MPI_Sendrecv`.

Exercise 2.3: π ***MPI Ring (point to point asynchronous)***

Modify the previous exercise to use `MPI_Isend` and `MPI_Recv`.

3 MPI collectives

Exercise 3.1: π ***MPI (collective gather)***

Instead of the ring to communicate a value between every process, use collective communication: Call `MPI_Gather` to collect all the partial sums in the root process. Then, through `MPI_Bcast`, broadcast the total sum to every process.

Exercise 3.2: π ***MPI (collective reduce)***

Modify the previous exercise to use `MPI_Reduce`.