Serie MPI

1 Different parallelisation of the Pi reduction

Exercise 1. MPI Hello world

- Initialize/finialize properly MPI
- Print out the number of processes and the rank of each process
- Write a batch script to run your parallel code

```
#!/bin/bash
#SBATCH -n <ntasks>
module purge
module load <mpi library>
srun <my_mpi_executable>
```

Note : *To use MPI on the cluster you first have to load a MPI implementation through the module* `mvapich2` *or* `intel-mpi`*. In addition in the SLURM environment you should use* `srun` *instead of* `mpiexec` *or* `mpirun`

Exercise 2. MPI Ring point to point synchronous

In this exercise every process will compute a portion of the integral. And then the partial sum will turn in ring in order for every process the be able to compute the full integral by summing all the partial sums.

- Split the sum space between the processes
- Implement a ring to communicate the partial sum between the processes. using `MPI_Ssend` and `MPI_Recv`

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

  Note : *in a loop the next process is* `(prank + 1) % psize` *and the previous is* `(prank - 1 + psize) % psize`

Exercise 3. MPI Ring point to point synchronous sendrecv

Modify the previous exercise to use `MPI_Sendrecv`

Exercise 4. MPI Ring point to point asynchronous

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

Exercise 5. MPI Ring collective gather

Instead of the ring to communicate a value between every process use the collective communication, `MPI_Gather` the partial sums to the root process. Then `MPI_Bcast` the total sum to everyone

Exercise 6. MPI Ring collective reduce

Modify the previous exercise to use `MPI_Reduce`

2 Poisson

Exercise 7. Parallelization with MPI

Parallelize the Poisson 2D problem using the Messages Passing Interface (MPI). As a starting point, you can use the debugged, profiled and optimized serial version of the serie 4 or start from scratch. Here follows some advises for your work:

- The memory allocation in C is done in "Row-Major Order" : make your domain decomposition by lines

  (In Fortran, the memory allocation is "Column-Major Order" (make the decomposition by columns))

- Try to keep the size of the MPI messages as large as possible (i.e. send/receive a full line instead of single elements). In order to avoid deadlocks, use `MPI_Sendrecv` first.

- Same problem as with OpenMP : the main bottleneck is the file writings: be sure not to call `dump()`. The verification is done by comparing the number of iterations to reach a given error (L_2). To be sure your parallel implementation is correct: compare your results against the serial implementation.

- To increase the performance of your code, the communications can be hidden behind computation by using non-blocking communications (`MPI_Isend`/`MPI_Irecv`)

- Increase the size of the grid so that the total execution time on one node is close to 3-4 seconds

- Run your application on an increasing number of nodes by fixing the total size of the problem. Draw a log-log graph with the speedup ($S_p = t1/tp$) on the y axis, the number of nodes on the x axis (**strong scaling**)

- Run your application on an increasing number of nodes by fixing the size of the problem per processor. Draw a graph with the parallel efficiency ($E_p = S_p/p$) on the y axis, the number of nodes on the x axis (**weak scaling**)

Exercise 8. Naive IOs

The first approach for the dump (`write_to_file`) function is to gather everything on the root node, usually the rank 0.

- For this exercise gather all the lines on the process of rank 0 and write the file on disk.

- Check the effect of this approach on the scalability.