Workshop 4: Parallelising the Lissajous program with MPI

In this workshop you will parallelise a version of the Lissajous program from workshop 2 using MPI (Message Passing Interface). This is an easier program to parallelise than the advection program because it is not necessary to exchange haloes between neighbouring sub-domains. However it is necessary to divide the work between the available MPI processes and to gather the results together so they can be written out.

The workshop is designed to be carried out using the Linux PCs in the Lovelace laboratory. It is assumed that you are not on campus and will access these computers remotely. On the module ELE page there are instructions describing how to log in to a remote Linux computer under the "Workshop 2" section.

1 Getting set up

- Start by logging in to a Lovelace lab Linux PC, using the Linux computer availability dashboard (`http://students.emps.ex.ac.uk/dashboard/`) to choose a computer where the system load is low (you may need to use the VPN to view the dashboard).
- The example programs for this workshop are provided as a tar file. Take a copy of the tar file for this workshop by running the command:

  ```
  cp /secamfs/userspace/ug/shared/ecm3446/workshop4.tar .
  ```

  where the dot at the end of the command causes the file to be copied into the current working directory.
- Unpack the tar file by running the following command:

  ```
  tar xvf workshop4.tar
  ```

  This will create a directory called `workshop4` which contains the example programs for this workshop.
- Load the `mpi` module so that you have access to the `mpicc` and `mpirun` commands:

  ```
  module load mpi
  ```

2 Parallelising the program

The program can be parallelised by dividing the points to be calculated between the available MPI processes so that each MPI process calculates a subset of the points. After the points have been calculated the results will be gathered onto the rank 0 process and written out to a file. This can be done without knowing at compile time how many processes will be used. We will assume that the number of points divides exactly by the number of MPI processes. The number of points in this version of the example program has 7560 points which divides exactly by integer values between 1 and 10 inclusive.

1. Start by compiling and running the serial version of the program to generate known good output.

```
cd workshop4
gcc -std=c99 -o lissajous lissajous.c -lm
./lissajous
```

Rename the output file so that it is not overwritten when you next run the program.

```
mv output.dat output_kgo.dat
```

2. To add MPI parallelism to the program follow these steps:

   (a) Start by including the `mpi.h` header file. Then add a call to `MPI_Init` after the variable declarations, and add a call to `MPI_Finalise` just before the `return` statement.

   (b) Add calls to `MPI_Comm_size` and `MPI_Comm_rank` to find out how many MPI processes there are and the rank of each process in the `MPI_COMM_WORLD` communicator.

   (c) Use the number of processes returned by `MPI_Comm_size` to work out how many points each MPI process needs to calculate (n). To do this divide the total number of points (N) by the number of MPI processes.

   (d) The number of points to be calculated by each process will not be known until run time. This means that don't know at compile time how large to make the arrays which hold the subset of the results for each individual process. Instead of declaring a static array we can allocate the memory required using `malloc` after the number of points per processor has been calculated. To do this:

   - Add declarations for two `float` pointers called `x_local` and `y_local` which will point to buffers holding the results from each process.
   - Add two calls to `malloc` to allocate enough memory to store `n` floats in the `x_local` and `y_local` buffers
   - Add two calls to `free` to free the memory immediately before the call to `MPI_Finalize`

   (e) Each process will calculate a subset of the points. Change the loop iterator so that the iterations run from `0` to `n-1` (i.e. number of points per process -1).

   (f) We need each MPI process to calculate a different range of values for the parameter `t`. Change the statement which calculates t so that

   $$t = 2\pi \frac{i + rn}{N} \tag{1}$$

   where n is the number of points per process, i is the loop iterator ($0 \leq i \leq n-1$), r is the process rank and N is the total number of points. This ensures that each point is calculated once, and only once, across all the MPI processes.

   (g) Change the loop body so that the results are stored in `x_local` and `y_local` instead of `x` and `y`.

   (h) After the loop make two calls to `MPI_Gather` to gather results from `x_local` and `y_local` into the `x` and `y` arrays on the rank 0 process.

   (i) Add an if statement so that only the rank 0 process writes the results out to file.

3. Compile the program using

   ```
   mpicc -std=c99 -o lissajous lissajous.c -lm
   ```

   then run using

   ```
   mpirun -np N ./lissajous
   ```

   where you should replace `N` with the number of MPI processes to use. Test the program using different numbers of MPI processes between 1 and 8, and check the output matches known good output from the serial version.