Things to Try in MPI

Rather than giving detailed problems, this sheet suggests some things to try based on the example code provided. All of these things are possibly using nothing beyond the slides and notes. Some are trickier than others!

- Variants of Correct Code

- There are several correct ways to write most code

Info:

The example folders contain some variants on the code discussed in the talks. Some of these use functions we mentioned but didn't discuss much.

Try:

When we built up the "Darts" code, we showed some different ways to do it; in particular the final data combining. Have a look at these versions and how they differ. Do they all make sense to you?

Compare the domain decomposition code using the manual splitting, to the one using the MPI built-ins (MPI_Cart_*) (parallel vs parallel_cart versions)

(Tricky) The Extras folder also contains a "Non-blocking" version of the ring-pass code. How does this work? What extra steps are needed to make sure everything stays "in sync"?

(Very tricky) Extras also contains a "semi-blocking" ring pass, where we mix non-blocking sends with blocking receives. Run this - how does it work? Run it on 2 processors - what slightly strange thing is happening (Hint:)

- Domain Decomposition

- We discussed the idea of minimising surface area (comms) to volume (calculation) - can we demonstrate this mattering in simple code?

Info:

Start with the code solving the heat equation. Make sure this compiles and runs.

The `time` utility (https://linux.die.net/man/1/time) is a handy way of seeing how long a program takes to run

Try:

Run the code without Display. This can be done in the Fortran by compiling as shown in the README. In C, use `make noio`

Run the code on two different numbers of processors, using `time`. Adjust the number of points to take a measurable amount of time. Does the runtime drop as expected?

(Tricky) Try re-adjusting the code to only split the domain in 1 direction. How inefficient is this? Notice the "run to run" variation in timing - anything comparable to this isn't going to be meaningful.

- Writing MPI code

- Adding MPI to a 1-D domain decomposed code
- Re-doing the simple dart program as a Worker-Controller

Info:

In domain decomposed codes it is easy to see whether your MPI is working correctly

The dart program is a good candidate for demonstrating a real worker-controller model

Try:

Try parallelising the 1-D version of the heat equation code. Use the manual rank calculations or the MPI_Cart_* as you prefer.

Based on the simple worker-controller model code, parallelise the dart program using fixed size work blocks (N iterations each). The result returned by the workers is the number of darts within the circle

(Tricky) adapt the program you made to use different sized work packages (different numbers of iterations). Keep track of the size given to each rank, rather than trying to return multiple values from the workers

(Very tricky) if you needed to use multiple messages to return the results, how might that work. (Hint:)

- Working to a Tolerance

  - Using MPI comms to control code

  Info:

  The heat equation code runs a fixed number of iterations. In real code we often want to work until we have reached some sort of goal.

  Try:

  Adapt your worker-controller darts program to run until the controller detects that you are "done". You can either (easy) pre-program a value for pi, and stop when you are close enough (say 1e-4) of it, or (tricky) stop when increasing the number of iterations changes the answer by less than some percentage.

  (Tricky) Adapt the heat equation code to run until the answer stops changing by more than x percent on each iteration. Important: do not let each processor determine this for itself! Your code will, sooner or later, deadlock. Can you explain why?

- Other Examples

  - Other models of MPI code

  Info:

  The Extras folder contains an example of a simple loop parallelised code

  Try:

  Have a look at the subloop code. What ensures the x-axis is distributed so that each processor has its own segment?

  What are the potential pitfalls of the final Reduce operation? Consider speed, memory requirements, and how it might behave on "many" (1000+) processors.