Parallel algorithms and programs
TP 2: Parallel sort

Jean-Marie Madiot, Julien Herrmann
jeanmarie.madiot@ens-lyon.fr,
julien.herrmann@ens-lyon.fr

Consider an array A of size n and $p = 2^k$ MPI processors to sort it.

Exercise 1

In a first phase, the array A is already distributed (for example, in p files each one of size n/p, but you can initialise it more simply with a pseudo-random generator, like RANDU[1]). There is no constraint on the total memory a processor can use. Each process must then allocate an array v of size n/p and read/copy the content of the file whose name is the process id in this array. Finally, each process call the function `sort` whose interface is:

```
void sort(double *v, int len, MPI_Comm comm);
```

with:

- `v`: the array v,
- `len`: the size of array v,
- `comm`: the MPI communicator, that contains all the processes involved in the sort operation.

1. Implement the merge sort algorithm `mergesort(double *v, int len)` to locally sort one array v.
2. Think of a smart parallel algorithm such that at the end, the virtual array consisting in the amalgamation of all v arrays of each processor by increasing MPI rank becomes sorted. What is the complexity of your algorithm? (Time? Communication? Memory?)
3. Implement this algorithm in function `sort1`.

Exercise 2

In a second phase we add two new constraints: processors are not able to use more than twice the memory necessary to store v, and we suppose we are in a ring topology.

1. Think of a modified parallel algorithm that takes into account this new constraint. What's the new complexity?
2. Implement this algorithm in function `sort2`, trying to optimise the total computation time, taking into account communication and I/O costs.

[1] RANDU is a very simple but poor generator, sadly used for many years.