Parallel computing

Exercise on basic communication primitives

- **Basic communication primitives**

  a) Programming models/libraries/languages such as BSP allow 'one-sided communication', i.e. a processor can write data in the memory of another processor, via the 'primitive' `bsp_put`.

  b) Programming models/libraries/languages such as (current) MPI library allow 'two-sided communication', where the basic communication primitives `send` and `receive` can be executed in several modes, incl.

  - *blocking non-buffered*: the communication only happens when the sending process has executed the `send` operation and the receiving process has executed the `receive` operation, e.g. a synchronisation between both processors is needed.
  - *blocking buffered*: in the sending process the `send` operation ends as soon as the message is copied into a system buffer (provided by the communication system); in the receiving process, the `receive` operation gets the message out of the system buffer, and thus it must perhaps wait until the message arrived in this buffer.

  We assume the following simple syntax for these basic communication primitives:

  - `bsp_put(source_pointer, dest_pointer, dest)`
  - `send (message, dest [, label])`
  - `receive (message, sender [, label])`

  `source_pointer:` *input* parameter: pointer to the 'source' memory location in the local processor from which the data are read

  `dest_pointer:` *input* parameter: pointer to the 'destination' memory location in the remote processo into which the data are written (Remark: to enable a processor to write in a remote variable, the 'local name' must be linked to the remote address. In BSP this is done via the registration primitive (to be described later), which involves communication.

  `dest:` *input* parameter : processor number (we assume 1 process per processor (=core))

  `message:` *input* parameter for `send` and `receive`
  pointer to the data structure where the message is stored/must be stored
  (also the length of the message is a parameter, but here we neglect this)

  `sender:` *output* parameter (hence in the receiving process one cannot select a message in the list of arrived messages based on the processor number of the sending processor)

  `label:` *input* parameter for `send`; *output* parameter for `receive`
  (this parameter may be absent)

  The following initialisation procedure is available

  `initcomm (myproc_id, p) :`
  `myproc_id :` output parameter : 'own processor number'
  `p :` output parameter : number of 'active' processors

  Hence, it is possible to write only one program, that will be executed on each processor, while nevertheless different parts of the algorithm will be executed on different processors, by using tests based on the processor number, for example

  `if (myproc_id is even) then …… endif`

  This approach is called the SPMD programming model (SPMD = Single Program, Multiple Data).

Exercise: exchange of information between 'neighbouring' processors

Assume a 'ring' of processors and an even number of processors – see figure.
Each processor must send the content of the variables `infoL` and `infoR` to respectively his left neighbour and his right neighbour; the message received by each processor from his left and right neighbour must be stored in resp. the variables `storeL` and `storeR`.

Write an algoritme in the SPMD programming model
a) using the 'one-sided communication' offered by `bsp_put`
b) using both execution modes of the 'two-sided communication' primitives