MPI – Part 4

1. In this task we ask you to sketch another version of *Homework 5, exercise 1* using one-sided communication. Notice that we are asking for a sketch (algorithm) and not actual C code. To solve this task, the following instructions are available:

   - `Y = fetch(var_name, j, label);`

     `fetch` gets the content of variable `var_name`, local to processor p_j, and stores it onto the local variable `Y`. `label` is a numeric label –local to p_i– that uniquely identifies the data transfer. The instruction is *one-sided* and *asynchronous*: p_i executes the next instruction in the program without waiting for `fetch` to complete; p_j does not participate in the data exchange.
   - `wait(label);`

     p_i idles until the data transfer identified by `label` is completed.
   - `barrier();`

     p_i idles until every processor issues a `barrier()`.

   Solution.

   All processes P_i have room for two blocks of size $\frac{n}{2} \times \frac{n}{2}$. We will assume two variables `blk0` and `blk1` point to those two blocks.

   The algorithm (see below) proceeds as follows. Each process initializes its block of A and stores it in `blk0` (line 2). Before starting the computation, the processes must synchronize (line 4) to make sure everyone initialized its data, so that subsequent data fetching from other processes contains the expected values.

   Next, each process gets the id of its partner (line 6), grabs the required data from its partner, waits until the data has been completely transfered, and operates (lines 8 through 10). Notice that they overwrite the block that was just transfered to avoid overwriting data that their partners may have not read yet. `op` corresponds to $+$ or $-$ depending on the process id.

   An explicit barrier (line 12) is then needed for two reasons:

   (a) Make sure the partner for the next stage has already computed its block of B, and data is ready to proceed with stage 2 and compute C.

   (b) Make sure that when reading the partner's block of B, we can already overwrite our local `blk0`. When passing through the barrier, our partner in stage 1 has already read and computed with the data in `blk0`.

   The second stage proceeds analogously. The final barrier synchronizes every process to be able to declare the job completed.

Algorithm 1 : Exercise 1. Computation of C with 4 processes.

```
1: # Initialize my blk0
2: blk0 = initialize(A, i)
3: # Sync to make sure everyone initialized
4: barrier()
5: # Get my partner's id for the first stage
6: mypartner = (i % 2 == 0) ? i+1 : i-1
7: # Grab, wait and operate
8: blk1 = fetch( blk0, mypartner, label0 )
9: wait( label0 )
10: blk1 = blk1 op blk0
11: # Sync to make sure everyone completed stage 1
12: barrier()
13: # Get my partner's id for the second stage
14: mypartner = (i / 2 == 0) ? i+2 : i-2
15: # Grab, wait and operate
16: blk0 = fetch( blk1, mypartner, label1 )
17: wait( label1 )
18: blk0 = blk0 op blk1
19: # Sync with everyone and we're done
20: barrier()
```