MI3.22 – Advanced Programming for HPC
Labwork 2 – *Multiplying matrix with Thrust*

This labwork investigates the parallel product of two square matrices. While this seems quite easy in the beautiful but so-slow sequential world, it becomes more difficult in the fast parallel galaxy. In practice, we need to apply some parallel patterns! Here, the difficulty grows with the exercises; so, do them in order.
NB: the size of matrices is $n \times n$; matrices are stored by row (from row 0 to row $n-1$).

Exercise 1: Summing two matrices

As appetizer, you have to implement the sum of two matrices. The goal is that you manipulate the `D_Matrix` structure, whose field `d_val` is a pointer on a pre-allocated **device** memory area. In others words, memory is handled for you already! The parallel pattern to use here is the *MAP* (NB: generally, underlined part is a HTTP link to Thrust documentation, that you should read!).

Exercise 2: Transposition

Implement the matrix transposition, using the *scatter* parallel pattern.
To know where to write the data, a solution is to use a `counting_iterator` modified using a `transform_iterator`. Basically, the counting iterator returns the index where input data is read. The transform iterator modifies this data to the index where to write. You have to use some combinatorial to transform the one-dimensional input index to a two-dimensional one, using division, modulo and the row size. Then you may permute the two coordinates (row, column), and rebuild a one-dimensional index.

Exercise 3: Diffusion

This function receives as input an integer i and a matrix M. It must return a new matrix that contains the i^{th}-row of M, repeated n times. Again, counting and transform iterators are useful, with just a modulo. The last iterator constructor must receive and set the parameters i and $M.d\_val$.
The parallel pattern to use is a simple *copy* (a kind of gather ... read the documentation!).

Exercise 4: First product

Your first implementation of the matrix product will use the following algorithm:

```
1  Function Product( A, B : D_Matrix ) : D_Matrix ;
2  Begin
3    D_Matrix TB := Transpose( B ) ; { Exercise 1 }
4    D_Matrix C( A.m_n ); { result is a n-times-n matrix }
5    FOR i=0 TO C.m_n - 1 DO { For each column of C }
6      D_Matrix D = Map(A,Diffusion(TB[i]),'*'); { Product of each line of A by the ith column of B }
7      { Reduction of each line of D, saved into vector ''Column'' }
8      SegmentedReduce( LineNumber, D, Column, '+' ) ;
9      Scatter( Column, ColumnElements(C, i) ) ; { Set the vector ''Column'' as the ith column of C }
10   End DO;
11   Product := C;
12 End Produit
```

At line 3, the transpose of B is made using exercise 1. At line 4, we just allocate the result matrix denoted C. Then, lines 5 and following, the C columns are calculated one-by-one: at first, line 6, we calculated matrix D as the "pseudo" product of A by a matrix containing the i^{th} column of B repeated n times (thanks to exercise 3) ; this product leads to a matrix for which each cell of index (i,j) is the product of two cells having same index both in A and `Diffusion(TB[i])`. Then, line 8 fills two vectors of size n, the first with the line number of D, the second with the reduction (using an addition as binary operator) made on each line (see *Reduce_By_Key*). The second resulting array contains the i^{th} column of C, that is scattered line 9. After the loop, it suffices to return the matrix C as the result (line 11).

Exercise 5: Second matrix product

Same question as for the previous exercise, but avoiding some useless memory transfer.

- To start, it is unnecessary to write the matrix D. Instead, you may use the iterator already used in exercise 3, directly.
- Then, in the segmented reduce it is possible to copy the obtained data (*id est* vector column) directly to a matrix named TC, for Transpose of C. This implies, line 11, to return the transpose of TC, and no more TC directly (since the transpose of the transpose of a matrix M is M).