# TP 5 : Derived Types

**Imad Kissami**

April 17, 2025

## Exercise 1 : Matrix Transposition using Derived Types

This exercise demonstrates how to use MPI derived types to transpose a matrix during communication. The source matrix is defined on process 0, and its transpose is received directly in memory on process 1 using advanced datatypes.

### Specifications

- A matrix $A$ of size $4 \times 5$ (rows $\times$ columns) is initialized on process 0.
- The objective is to send the matrix from process 0 to process 1, such that process 1 receives the **transposed matrix $A^T$** of size $5 \times 4$ directly in memory.
- The transposition must be performed using MPI derived datatypes :
  - `MPI_Type_vector` to define a column layout.
  - `MPI_Type_create_hvector` to build the full transpose structure in memory.
  - A single call to `MPI_Send` and `MPI_Recv`.

### Instructions

1. Define a matrix `a[4][5]` on process 0 and fill it with values from 1 to 20.
2. Display matrix `a` on process 0 before sending.
3. On process 1, declare a matrix `at[5][4]` to hold the transposed result.
4. Build an adequate derived datatype
5. Commit the type and use it in a single `MPI_Recv` call on process 1.
6. Send the original matrix from process 0 with a regular `MPI_Send`.
7. Display matrix `at` on process 1 after reception.

### Expected Output

```
Process 0 - Matrix a:
1     2     3     4     5
6     7     8     9    10
11    12    13    14    15
16    17    18    19    20

Process 1 - Matrix transposee at:
1     6    11    16
2     7    12    17
3     8    13    18
4     9    14    19
5    10    15    20
```

## Exercise 2 : Distributed Gradient Descent with MPI Derived Types

This exercise aims to implement a distributed version of the batch gradient descent algorithm using MPI. You will apply 'MPI_Type_create_struct' to define a custom MPI datatype for exchanging training samples (features + label) across processes.

### Context

Each training sample contains :

— A feature vector `x[N_FEATURES]` (e.g. 5 features),

— A scalar label `y`.

All samples are stored as a structure :

```
typedef struct {
 double x[N_FEATURES];
 double y;
} Sample;
```

### Goal

Parallelize the gradient descent algorithm (`distrib_grad.c`) over multiple MPI processes. Each process will :

— Compute a local gradient and local loss (MSE),

— Update the weight vector synchronously on all processes,

— Stop when the MSE is below a fixed threshold.

### Instructions

1. Define a derived type using `MPI_Type_create_struct` for the `Sample` struct.

2. Generate the full dataset on process 0 using the provided function `generate_data()`.
3. Scatter the dataset to all processes using `MPI_Scatterv` with the derived type.
4. Each process computes its local loss and gradient.
5. Aggregate gradients and losses.
6. All processes update their local copy of the weight vector.
7. Print loss and weight updates every 10 epochs from process 0.
8. Stop early if the global loss becomes smaller than the predefined threshold.
9. Increase the number of samples and plot the speedup and Efficiency using Toubkal (1 to 56 cores).

## Expected Output (Example)

```
Epoch 10 | Loss (MSE): 3.556296 | w[0]: -0.7849, w[1]: 0.5249
Epoch 20 | Loss (MSE): 3.326439 | w[0]: -0.6931, w[1]: 0.4749
...
Epoch 930 | Loss (MSE): 0.010897 | w[0]: 1.8730, w[1]: -0.9289
Epoch 940 | Loss (MSE): 0.010408 | w[0]: 1.8773, w[1]: -0.9312
Early stopping at epoch 949 - loss 0.009995 < 1.0e-02

Training time: 3.529 seconds (MPI)
```