Wednesday, 12/21/11:

Representation of Matrices in C:

**Do not use \( A[i][j] \) in C; use \( A[i][j] \) or \( A[i][j] \) instead.**

```
double A[10][10] or
double A[n][n]
```

How are the data actually organized?

**Dynamic way:**

```
double **A
A = (double**) malloc (n * sizeof (**double*))
for (i = 0; i < n; i++)
A[i] = (**double) malloc (n * sizeof (double))
```

=> elements \( A[i][j] = A_{ij} \)

=> the malloc's are all independent; no data are not consecutive in memory?

The same is true for \( A[n][n] \).

Notice: C organizes matrices by row = row-major ordering. This is incompatible with Fortran and Matlab and thus also all major numerical libraries. More subtly, the double pointed in \( A[n][n] \) is a problem since in other languages passing arrays in this way \( A[i][j] \) is simply not the same as \( A[i][j] \).
Therefore, one 1-D array for all arrays including 2-D matrices or higher dimensional arrays and then index yourself manually.

double *A

A = (double*) malloc (n*n * sizeof (double))

Use it as \( A[i+n*j] = A_{ij} \) (or \( A[i+n*n+j*n] = A_{ij} \))

This is column-major ordering, since \( A[i+n*j] \) is next to \( A[i+n*j+1] \)

in memory, which is \( A_{ij} \) being next to \( A_{ij+1} \).

This way we can have complete control of where our data are.

This is vital to using MPI communication commands and to efficient code.

Example:

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}
\]

We divide into blocks of columns to accommodate the column-major ordering,

since then all the \( A_{ij} \) are consecutive in memory.

Now we can do a single MPI_Gather to collect all \( E_A \) into \( A \)
on Process 0 for fast output \((-H90\text{ (a)})\)

\text{MPI-Gather} (E_A, r*j, MPI_DOUBLE, A, n*n, MPI_DOUBLE, \ O, MOC.COMM_WORLD)

 Clarify:

\[
j = i + j + id * l_n
\]

\[
l_A = allocate\_double\_vector (n * l_n)
\]

\[
\text{for} \ (i = 0; i < n; i++)
\]

\[
\text{for} \ (j = 0; j < l_n; j++)
\]

\[
l_A[i + n * l_j] = A_{ij}
\]