frovedis::sliced_blockcyclic_matrix<T>

NAME

`frovedis::sliced_blockcyclic_matrix<T>` - a data structure containing the slicing information of a two-dimensional `frovedis::blockcyclic_matrix<T>`

SYNOPSIS

```
#include <frovedis/matrix/sliced_matrix.hpp>
```

Constructors

sliced_blockcyclic_matrix ()
sliced_blockcyclic_matrix (const `blockcyclic_matrix<T>`& m)

Public Member Functions

void set_num (size_t nrow, size_t ncol)

Public Data Members

`node_local<sliced_blockcyclic_matrix_local<T>>` data
size_t num_row
size_t num_col

DESCRIPTION

In order to perform matrix operations on sub-matrices instead of entire physical matrix, frovedis provides some sliced data structures. `sliced_blockcyclic_matrix<T>` is one of them. It is actually not a real matrix, rather it only contains some slicing information of a physical `blockcyclic_matrix<T>`. Thus any changes performed on the sliced matrix, would actually make changes on the physical matrix from which slice was made.

Like `blockcylic_matrix<T>`, a `sliced_blockcyclic_matrix<T>` is also a template based structure with type **"T"** which can be either **"float"** or **"double"** (at this moment). Specifying other types can cause floating point exception issues.

A `sliced_blockcyclic_matrix<T>` contains public member "data" of the type `node_local<sliced_blockcyclic_matrix_local<T>>`. The actual distributed sliced matrices are contained in all the worker nodes locally, thus named as `sliced_blockcyclic_matrix_local<T>`. Each of this local matrix has the below structure:

```
template <class T>
struct sliced_blockcyclic_matrix_local {
  T *data;  // pointer to the beginning of the physical local matrix
  int *descA; // pointer to the descriptor array of the physical local matrix
  size_t IA;  // row-id of the physical matrix starting row from which to slice
  size_t JA;  // col-id of the physical matrix starting col from which to slice
  size_t sliced_num_row;  // number of rows in the global sliced matrix
  size_t sliced_num_col;  // number of columns in the global sliced matrix
};
```

E.g., if a physical `blockcyclic_matrix<T>` M has dimension 4x4 and slice is needed from 2nd row and 2nd column till 3rd row and 3rd column, then
"data" in each node will hold the address of local blockcyclic matrix of that node (data -> &local_m[0][0]),
"descA" in each node will hold the address of the array descriptor of the local blockcyclic matrix of that node (descA -> &local_m.descA[0]),
"IA" will be 2 (starting from 2nd row - so row id is 2),
"JA" will be 2 (starting from 2nd column - so col id is 2),
From 2nd row till 3rd row, number of rows to be sliced is 2, thus "sliced_num_row" would be 2.
From 2nd column till 3rd column, number of columns to be sliced is 2, thus "sliced_num_col" would be 2.

Kindly note that IA and JA do not contain the index, instead they contain the id. And also in each local sliced matrices the sliced information IA, JA, sliced_num_row, sliced_num_col would be same. The only difference would be in the pointer values, i.e., in data and descA.

The global version, `sliced_blockcyclic_matrix<T>` at master node actually contains nothing but the reference to these local sliced matrices at worker nodes and the global matrix dimension, i.e., the actual number of rows and columns in the physical global blockcyclic matrix. It has the below structure:

```
template <class T>
struct sliced_blockcyclic_matrix {
  node_local<sliced_blockcyclic_matrix_local<T>> data; // local matrix information
  size_t num_row;   // actual number of rows in physical global matrix
  size_t num_col;   // actual number of columns in physical global matrix
};
```

Such matrices are very useful in operations of external libraries like pblas, scalapack etc.

Constructor Documentation

sliced_blockcyclic_matrix ()

Default constructor. It creates an empty sliced matrix with num_row = num_col = 0 and local data pointers pointing to NULL. Basically of no use, unless it is needed to manipulate the slice information manually.

sliced_blockcyclic_matrix (const `blockcyclic_matrix<T>`& m)

Special constructor for implicit conversion. This constructor treats an entire physical matrix as a sliced matrix. Thus the created `sliced_blockcyclic_matrix<T>` would have same dimension as with the input `blockcyclic_matrix<T>` and local data pointers pointing to the base address of the local blockcyclic matrices.

Public Member Function Documentation

void set_num (size_t nrow, size_t ncol)

This function sets the "num_row" and "num_col" fields with the actual number of rows and columns in the global physical blockcyclic matrix. Only useful when manual manipulation is required.

Public Data Member Documentation

data

An instance of `node_local<sliced_blockcyclic_matrix_local<T>>` which contains the references to the local sliced matrices in the worker nodes.

num_row

A size_t attribute to contain the actual number of rows in the physical global blockcyclic matrix.

num_col

A size_t attribute to contain the actual number of columns in the physical global blockcyclic matrix.

Public Global Function Documentation

make_sliced_blockcyclic_matrix ()

This utility function accepts a valid `sliced_blockcyclic_matrix<T>` and slicing information like row and column index from which slicing is to be started, and the size of the output sliced matrix, i.e., number of rows and columns to be sliced from the starting location. On receiving the valid inputs, it outputs a `sliced_blockcyclic_matrix<T>` object containing the reference to the local sliced matrices, else it throws an exception. It has the below syntax:

```
sliced_blockcyclic_matrix<T>
make_sliced_blockcyclic_matrix (const sliced_blockcyclic_matrix<T>& mat,
                                size_t start_row_index,
                                size_t start_col_index,
                                size_t num_row,
                                size_t num_col);
```

Please note that in case a `blockcyclic_matrix<T>` is passed to this function, the entire matrix would be treated as a `sliced_blockcyclic_matrix<T>` because of the implicit conversion constructor (explained above). Thus this function can be used to obtain a sliced matrix from both a physical `blockcyclic_matrix<T>` and a valid `sliced_blockcyclic_matrix<T>`.

Example: If a physical `blockcyclic_matrix<T>` "mat" has the dimension 4x4 and slicing is required from its 2nd row and 2nd column till 4th row and 4th column, then this function should be called like:

```
auto smat = make_sliced_blockcyclic_matrix(mat,1,1,3,3);
```

Index of the 2nd row is 1, thus start_row_index = 1
Index of the 2nd column is 1, thus start_col_index = 1
From 2nd row till 4th row, number of rows to be sliced is 3, thus num_row = 3
From 2nd column till 4th column, number of columns to be sliced is 3, thus num_col = 3

```
Input (mat):          Output (smat):
------------          --------------
1 2 3 4               6 7 8
5 6 7 8       =>      7 6 5
8 7 6 5               3 2 1
4 3 2 1
```

Now if we need to slice further this sliced matrix, "smat" from its 2nd row and 2nd column till 3rd row and 3rd column, then we would call this function like below:

```
auso ssmat = make_sliced_blockcyclic_matrix(smat,1,1,2,2);
```

Index of the 2nd row of smat is 1, thus start_row_index = 1
Index of the 2nd column of smat is 1, thus start_col_index = 1
From 2nd row till 3rd row of smat, number of rows to be sliced is 2, thus num_row = 2
From 2nd column till 3rd column of smat, number of columns to be sliced is 2, thus num_col = 2

Kindly note that 2nd row of "smat" is actually the 3rd row of the physical matrix "mat", but this function takes care of it internally. Thus you just need to take care of the index of the input sliced matrix, not the actual physical matrix

```
Input (smat):         Output (ssmat):
-------------         ---------------
6 7 8                 6 5
7 6 5          =>     2 1
3 2 1
```

The above input/output is presented just to explain the slicing concept. The internal storage representation of these sliced blockcyclic matrices would be a bit different and complex in nature.

SEE ALSO

blockcyclic_matrix, sliced_blockcyclic_vector