frovedis::sliced_blockcyclic_vector<T>

NAME

`frovedis::sliced_blockcyclic_vector<T>` - a data structure containing the row or column wise slicing information of a two-dimensional `frovedis::blockcyclic_matrix<T>`

SYNOPSIS

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

Constructors

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

Public Member Functions

void set_num (size_t len)

Public Data Members

`node_local<sliced_blockcyclic_vector_local<T>>` data
size_t size

DESCRIPTION

In order to perform vector operations on some rows or on some columns of a dense matrix, frovedis provides some sliced data structures. `sliced_blockcyclic_vector<T>` is one of them. It is actually not a real vector, rather it only contains some slicing information of a physical `blockcyclic_matrix<T>`. Thus any changes performed on the sliced vector, would actually make changes on the specific row or column of the physical matrix from which the slice was made.

Like `blockcyclic_matrix<T>`, a `sliced_blockcyclic_vector<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_vector<T>` contains public member "data" of the type `node_local<sliced_blockcyclic_vector_local<T>>`. The actual distributed sliced vectors are contained in all the worker nodes locally, thus named as `sliced_blockcyclic_vector_local<T>`. Each of this local vector has the below structure:

```
template <class T>
struct sliced_blockcyclic_vector_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 size;    // number of elements in the sliced vector
  size_t stride;  // stride between two consecutive elements of the sliced vector
};
```

E.g., if a physical `blockcyclic_matrix<T>` M has dimension 4x4 and its 2nd row needs to be sliced, 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 (row id of the 2nd row is 2),
"JA" will be 1 (since complete 2nd row needs to be sliced, column id of the first element in the 2nd row would be 1),
"size" will be 4 (number of elements in 2nd row),
"stride" will be 4 (since matrix is stored in colmajor order, the stride between two consecutive elements in a row would be equal to leading dimension of that matrix, i.e., number of rows in that matrix)

On the other hand, if 2nd column needs to be sliced, 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 1 (since complete 2nd column needs to be sliced, row id of the first element in the 2nd column would be 1),
"JA" will be 2 (column id of the 2nd column is 2),
"size" will be 4 (number of elements in 2nd column),
"stride" will be 1 (since matrix is stored in colmajor order, the consecutive elements in a column would be placed one after another)

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

The global version, `sliced_blockcyclic_vector<T>` at master node actually contains nothing but the reference to these local sliced vectors at worker nodes and the global size of the distributed row or column vector. It has the below structure:

```
template <class T>
struct sliced_blockcyclic_vector {
  node_local<sliced_blockcyclic_vector_local<T>> data; // local vector information
  size_t size;  // actual no. of elements in the target row/col in the physical matrix
};
```

Such vectors are very useful in operations of external libraries like pblas etc.

Constructor Documentation

sliced_blockcyclic_vector ()

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

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

Special constructor for implicit conversion. This constructor treats an entire physical matrix as a sliced vector. Thus the created `sliced_blockcyclic_vector<T>` would have "size" equals to number of rows in the input `blockcyclic_matrix<T>` and "data" pointer pointing to the base address of the local blockcyclic matrices. Please note that such conversion can only be posible if the input matrix can be treated as a column vector (a matrix with multiple rows and single column), else it throws an exception.

Public Member Function Documentation

void set_num (size_t len)

This function sets the "size" field with the actual number of elements in the target row or column 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_vector_local<T>>` which contains the references to the local sliced vectors in the worker nodes.

size

A size_t attribute to contain the actual number of elements in the target row or column in the physical global blockcyclic matrix.

Public Global Function Documentation

make_row_vector ()

This utility function accepts a valid `sliced_blockcyclic_matrix<T>` and the row index to be sliced. On receiving the valid inputs, it outputs a `sliced_blockcyclic_vector<T>` object containing the reference to the local sliced vectors, else it throws an exception. It has the below syntax:

```
sliced_blockcyclic_vector<T>
make_row_vector (const sliced_blockcyclic_matrix<T>& mat,
                 size_t row_index);
```

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 (as explained in *sliced_blockcyclic_matrix* manual page). Thus this function can be used to obtain a row vector 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 its 2nd row needs to be sliced, then this function should be called like:

```
auto rvec = make_row_vector(mat,1); // row index of second row is 1

Input (mat):        Output (rvec):
```

```
------------        --------------
 1 2 3 4       =>    5 6 7 8
 5 6 7 8
 8 7 6 5
 4 3 2 1
```

Now if it is needed to slice the 2nd row from its 4th block (sub-matrix), then the operations can be performed as per the code below:

```
auto smat   = make_sliced_blockcyclic_matrix(mat,2,2,2,2);
auto s_rvec = make_row_vector(smat,1);
```

First the original matrix needs to be sliced to get its 4th block (3rd row and 3rd column till 4th row and 4th column) and then 2nd row is to be sliced from the sub-matrix.

Kindly note that 2nd row of "smat" is actually the 4th 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 (mat):       Output (smat):      Output (s_rvec):
-------------      --------------      ----------------
 1 2 3 4            6 5            =>   2 1
 5 6 7 8       =>   2 1
 8 7 6 5
 4 3 2 1
```

make_col_vector ()

This utility function accepts a valid `sliced_blockcyclic_matrix<T>` and the column index to be sliced. On receiving the valid inputs, it outputs a `sliced_blockcyclic_vector<T>` object containing the reference to the local sliced vectors, else it throws an exception. It has the below syntax:

```
sliced_blockcyclic_vector<T>
make_col_vector (const sliced_blockcyclic_matrix<T>& mat,
                 size_t col_index);
```

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 (as explained in *sliced_blockcyclic_matrix* manual page). Thus this function can be used to obtain a column vector 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 its 2nd column needs to be sliced, then this function should be called like:

```
auto cvec = make_col_vector(mat,1); // col index of second col is 1
```

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

Now if it is needed to slice the 2nd column from its 4th block (sub-matrix), then the operations can be performed as per the code below:

```
auto smat   = make_sliced_blockcyclic_matrix(mat,2,2,2,2);
auto s_cvec = make_col_vector(smat,1);
```

First the original matrix needs to be sliced to get its 4th block (3rd row and 3rd column till 4th row and 4th column) and then 2nd column is to be sliced from the sub-matrix.

Kindly note that 2nd column of "smat" is actually the 4th column 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 (mat):       Output (smat):      Output (s_cvec):
-------------      --------------      ----------------
1 2 3 4            6 5             =>  5 1
5 6 7 8       =>   2 1
8 7 6 5
4 3 2 1
```

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

SEE ALSO

blockcyclic_matrix, sliced_blockcyclic_matrix