frovedis::node_local<T>

NAME

`frovedis::node_local<T>` - a distributed object of type 'T' stored locally at each worker nodes

SYNOPSIS

```
#include <frovedis.hpp>
```

Constructors

node_local ()
node_local (const `node_local<T>`& src)
node_local (`node_local<T>`&& src)

Overloaded Operators

`node_local<T>`& operator= (const `node_local<T>`& src)
`node_local<T>`& operator= (`node_local<T>`&& src)

Public Member Functions

```
template <class R, class F>
node_local<R> map(const F& f);
```

```
template <class R, class U, class F>
node_local<R> map(const F& f, const node_local<U>& l);
```

```
template <class R, class U, class V, class F>
node_local<R> map(const F& f, const node_local<U>& l1,
       const node_local<V>& l2);
```

```
template <class R, class U, class V, class W, class F>
node_local<R> map(const F& f, const node_local<U>& l1,
       const node_local<V>& l2, const node_local<W>& l3);
```

```
template <class R, class U, class V, class W, class X, class F>
node_local<R> map(const F& f, const node_local<U>& l1,
       const node_local<V>& l2, const node_local<W>& l3,
       const node_local<X>& l4);
```

```
template <class R, class U, class V, class W, class X, class Y, class F>
node_local<R> map(const F& f, const node_local<U>& l1,
```

```
       const node_local<V>& l2, const node_local<W>& l3,
       const node_local<X>& l4, const node_local<Y>& l5);

template <class R, class TT>
node_local<R> map(R(*f)(TT));

template <class R, class U, class TT, class UU>
node_local<R> map(R(*f)(TT, UU), const node_local<U>& l);

template <class R, class U, class V, class TT, class UU, class VV>
node_local<R> map(R(*f)(TT, UU, VV), const node_local<U>& l1,
       const node_local<V>& l2);

template <class R, class U, class V, class W,
     class TT, class UU, class VV, class WW>
node_local<R> map(R(*f)(TT, UU, VV, WW), const node_local<U>& l1,
       const node_local<V>& l2, const node_local<W>& l3);

template <class R, class U, class V, class W, class X,
     class TT, class UU, class VV, class WW, class XX>
node_local<R> map(R(*f)(TT, UU, VV, WW, XX), const node_local<U>& l1,
       const node_local<V>& l2, const node_local<W>& l3,
       const node_local<X>& l4);

template <class R, class U, class V, class W, class X, class Y,
     class TT, class UU, class VV, class WW, class XX, class YY>
node_local<R> map(R(*f)(TT, UU, VV, WW, XX, YY), const node_local<U>& l1,
       const node_local<V>& l2, const node_local<W>& l3,
       const node_local<X>& l4, const node_local<Y>& l5);

template <class F>
node_local<T>& mapv(const F& f);

template <class U, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l);

template <class U, class V, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l1,
       const node_local<V>& l2);

template <class U, class V, class W, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l1,
       const node_local<V>& l2, const node_local<W>& l3);

template <class U, class V, class W, class X, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l1,
       const node_local<V>& l2, const node_local<W>& l3,
       const node_local<X>& l4);

template <class U, class V, class W, class X, class Y, class F>
node_local<T>& mapv(const F& f, const node_local<U>& l1,
       const node_local<V>& l2, const node_local<W>& l3,
       const node_local<X>& l4, const node_local<Y>& l5);

template <class TT>
node_local<T>& mapv(void(*f)(TT));

template <class U, class TT, class UU>
node_local<T>& mapv(void(*f)(TT,UU), const node_local<U>& l);

template <class U, class V, class TT, class UU, class VV>
node_local<T>& mapv(void(*f)(TT, UU, VV), const node_local<U>& l1,
       const node_local<V>& l2);
```

template <class U, class V, class W,
class TT, class UU, class VV, class WW>
node_local<T>& mapv(void(*f)(TT, UU, VV, WW), const node_local<U>& l1,
const node_local<V>& l2, const node_local<W>& l3);

template <class U, class V, class W, class X,
class TT, class UU, class VV, class WW, class XX>
node_local<T>& mapv(void(*f)(TT, UU, VV, WW, XX), const node_local<U>& l1,
const node_local<V>& l2, const node_local<W>& l3,
const node_local<X>& l4);

template <class U, class V, class W, class X, class Y,
class TT, class UU, class VV, class WW, class XX, class YY>
node_local<T>& mapv(void(*f)(TT, UU, VV, WW, XX, YY), const node_local<U>& l1,
const node_local<V>& l2, const node_local<W>& l3,
const node_local<X>& l4, const node_local<Y>& l5);

template <class F> T reduce(const F& f);
template <class TT, class UU> T reduce(T(*f)(TT,UU));

template <class F> node_local<T> allreduce(const F& f);
template <class TT, class UU> node_local<T> allreduce(T(*f)(TT,UU));

std::vector<T> gather();
T vector_sum();
void put(int n_id, const T& val);
T get(int n_id);

template <class U> dvector<U> as_dvector() const;
template <class U> dvector<U> moveto_dvector();
template <class U> dvector<U> viewas_dvector();

DESCRIPTION

Frovedis provides an efficient data structure to perform an operation locally on a distributed data either broadcasted or scattered. When a data of type "T" is broadcasted or a vector containing elements of type "vector" is scattered among worker nodes, a node local view of those data can be represented by a `node_local<T>` or a `node_local<std::vector<T>>` object respectively.

Let's consider there are two worker nodes and an integer object containing "5" is broadcasted to them and a vector containing {{1,2},{3,4}} is scattered to the participating worker nodes. Then a node local view of these data can be picturized as below:

```
iData(5) -> broadcast
iVector({{1,2},{3,4}}) -> scatter

master       worker0     worker1
-----        -----       -----
d_iData      (5)         (5)
d_iVector    ({1,2})     ({3,4})
```

The d_iData and d_iVector in the above case can be considered as `node_local<int>` and `node_local<std::vector<int>>` respectively. These will provide the local view of the distributed data allowing user to perform the operations locally on each worker node in a faster and efficient way.

Such kind of data structure is useful in many machine learning algorithms, where the training process can be performed on the training data stored locally at the worker nodes in parallel and then reducing the local model to update the global model at master node etc.

Since the node_local provides a local view of the distributed object, a user is supposed to define the operation to be performed on each worker data (in case of a scattered vector, operation needs to be defined on each local vectors, instead of each elements like in dvector) in a map() like call. The next section explains functionalities of node_local in details.

Constructor Documentation

node_local ()

This is the default constructor which creates an empty node_local object. But it does not allocate any memory for the container. See make_node_local_allocate().

node_local (const node_local<T>& src)

This is the copy constructor which creates a new node_local of type T by copying the distributed data from the input node_local object.

node_local (node_local<T>&& src)

This is the move constructor. Instead of copying the input rvalue node_local, it attempts to move the contents to the newly constructed node_local object. It is faster and recommended when input node_local object will no longer be needed.

Overloaded Operator Documentation

node_local<T>& operator= (const node_local<T>& src)

It copies the source node_local object into the left-hand side target node_local object of the assignment operator "=". After successful copying, it returns the reference of the target node_local object.

node_local<T>& operator= (node_local<T>&& src)

Instead of copying, it moves the contents of the source rvalue node_local object into the left-hand side target node_local object of the assignment operator "=". It is faster and recommended when source node_local object will no longer be needed. It returns the reference of the target node_local object after the successful assignment operation.

Public Member Function Documentation

map()

The map() function is used to specify the target operation to be mapped on each worker data (each node local partition) of the distributed object. It accepts a function or a function object (functor) and applies the same to each worker data in parallel. Then a new node_local object is created from the return value of the function.

Along with the function argument, map() can accept maximum of five distributed data of node_local type. This section will explain them in details.

```
node_local<R> map(R(*f)(TT));
```

Below are the points to be noted while using the above map() interface.

- it accepts only the function to be mapped as an argument.
- thus the input function must also not accept more than one arguments.
- the type of the function argument must be same or compatible with the type of the node_local object.
- the return type, R can be anything. The resultant node_local object will be of the same type.

For example,

```
float func1 (float x) { return 2*x; }
float func2 (double x) { return 2*x; }
float func3 (other_type x) { return 2*x.val; }
double func4 (float x) { return 2*x; }

// let's consider "nloc" is a node_local of type float
// nloc is node_local<float>, func1() accepts float.
auto r1 = nloc.map(func1); // Ok, r1 would be node_local<float>.

// nloc is node_local<float>, func2() accepts double.
// but float is compatible with double.
auto r2 = nloc.map(func2); // Ok, r2 would be node_local<float>.

// nloc is node_local<float>, but func3() accepts some user type (other_type).
// even if the member "val" of "other_type" is of float type,
// it will be an error.
auto r3 = nloc.map(func3); // error

// func4() accepts float (ok) and returs double,
// but no problem with return type.
auto r4 = nloc.map(func4); // Ok, r4 would be node_local<double>.

// it is possible to chain the map calls
auto r5 = nloc.map(func1).map(func4); // Ok, r5 would be node_local<double>.
```

In the above case, functions accepting only one argument would be allowed to pass. If more than one arguments are to be passed, different version of map() interface needs to be used. Frovedis supports map() interface which can accept a function with maximum of five arguments as follows.

```
node_local<R> map(R(*f)(TT, UU, VV, WW, XX, YY), const node_local<U>& l1,
                const node_local<V>& l2, const node_local<W>& l3,
                const node_local<X>& l4, const node_local<Y>& l5);
```

When using the map() interface accepting function to be mapped with more than one arguments, the below points are to be noted.

- the first argument of the map interface must be the function pointer to be mapped on the target node_local.
- the type of the node_local and the type of the first function argument must be of the same or of compatible type.

- the other arguments of the map (apart from the function pointer) must be of distributed `node_local<T>` type, where "T" can be of any type and the corresponding function arguments should be of the same type.
- the return type, R can be anything. The resultant node_local object will be of the same type.

The mapping of the argument types of the map() call and the argument types of the function to be mapped on a node_local, "nloc" will be as follows:

```
func(d,x1,x2,x3,x4,x5);      nloc.map(func,l1,l2,l3,l4,l5);
---------------------        ---------------------
   d: T                         nloc: node_local<T>
   x1: U                        l1: node_local<U>
   x2: V                        l2: node_local<V>
   x3: W                        l3: node_local<W>
   x4: X                        l4: node_local<X>
   x5: Y                        l5: node_local<Y>
```

For example,

```
std::vector<int> func1(const std::vector<int>& x, int y) {
   std::vector<int> ret(x.size());
   for(auto i=0; i<x.size(); ++i) ret[i] = x[i] + y;
   return ret;
}
std::vector<double> func2(const std::vector<int>& x,
                          float y, double z) {
   std::vector<double> ret(x.size());
   for(auto i=0; i<x.size(); ++i) ret[i] = x[i] * y + z;
   return ret;
}

// let's consider "nloc" is a node_local of type "std::vector<int>"
// nloc is node_local<vector<int>> and func1() accepts
// "vector<int>" as first argument. (Ok)
// But second argument of the map() is simply "int" type in the below call,
// thus it will lead to an error.
auto r1 = nloc.map(func1, 2); // error

// broadcasting integer "y" to all workers to obtain node_local<int>.
int y = 2;
auto dy = broadcast(y);
auto r2 = nloc.map(func1, dy); // Ok, r2 would be node_local<vector<int>>

float y = 2.0;
double z = 3.1;
auto dy = broadcast(y); // dy is node_local<float>
auto dz = broadcast(z); // dz is node_local<double>
auto r3 = nloc.map(func2, dy, dz); // Ok, r3 would be node_local<vector<double>>
```

Thus there are limitations on map() interface. It can not accept more than five distributed parameters. And also all of the parameters (except function pointer) have to be distributed before calling map (can not pass non-distributed parameter).

These limitations of map() can be addressed with the map() interfaces accepting functor (function object), instead of function pointer. This section will explain them in details.

Below are the points to be noted when passing a functor (function object) in calling the map() function.

- the first argument of the map() interface must be a functor definition.

  node_local map(const F& f);

- the type of the node_local must be same or compatible with the type of the first argument of the overloaded "operator()" of the functor.

- apart from the functor, the map() interface can accept a maximum of five distributed node_local objects of any type as follows.

  node_local map(const F& f, const node_local& l1,
  const node_local& l2, const node_local& l3,
  const node_local& l4, const node_local& l5);

Where U, V, W, X, Y can be of any type and the corresponding arguments of the overloaded "operator()" must be of the same or compatible type.

- the functor itself can have any number of data members of any type and they need not to be of the distributed type and they must be specified with "SERIALIZE" macro. If the functor does not have any data members, then the "struct" definition must be ended with "SERIALIZE_NONE" macro.

- the return type, R of the overloaded "operator()", can be anything. The resultant node_local would be of the same type. But the type needs to be explicitly defined while calling the map() interface.

For example,

```
struct foo {
  foo() {}
  foo(float a, float b): al(a), be(b) {}
  std::vector<double> operator() (std::vector<int>& x) { // 1st definition
    std::vector<double> ret(x.size());
    for(auto i=0; i<x.size(); ++i) ret[i] = al*x[i]+be;
    return ret;
  }
  std::vector<double> operator() (std::vector<int>& x, int y) { // 2nd definition
    std::vector<double> ret(x.size());
    for(auto i=0; i<x.size(); ++i) ret[i] = al*x[i]+be*y;
    return ret;
  }
  float al, be;
  SERIALIZE(al,be)
};

// let's consider "nloc" is a node_local of "std::vector<int>" type.
// the below call will be ok, r1 would be node_local<vector<double>>
auto r1 = nloc.map<vector<double>>(foo(2.0,3.0));
```

In the above call of map(), it is taking a function object with "al" and "be" values as 2.0 and 3.0 respectively. Since these are the values for initializing the members of the function object, they can be passed like a simple constructor call.

"nloc" is `node_local<vector<int>>` and map() is called with only functor definition. Thus it will hit the first definition of the overloaded "operator()". The return type is `std::vector<double>` which can be of any type and needs to be explicitly mentioned while calling the map() function like `map<vector<double>>()` (otherwise some compiler errors might be encountered).

Like map() with function pointer, map with function object can also accept up to five distributed node_local objects of any type.

For example, in order to hit the 2nd definition of the overloaded "operator()" in previous foo structure, the map() function can be called as follows:

```
int be = 2;
// "be" needs to be broadcasted to all workers before calling the below
// map() function in order to get node_local<int> object. r2 would be
// node_local<vector<double>>.
auto r2 = nloc.map<vector<double>>(foo(2.0,3.0),broadcast(be));
```

Using function object is a bit faster than using a function, because it can be inline-expanded. On SX, it might become much faster, because in the case of function pointer, the loop cannot be vectorized, but using function object makes it possible to vectorize the loop.

Note mapping a function on a `node_local<vector<T>>` is equivalent to perform map_partitions() on a `dvector<T>`.

mapv()

The mapv() function is also used to specify the target operation to be mapped on each element of the node_local. It accepts a void returning function or a function object (functor) and applies the same to each worker data in parallel.
Since the applied function does not return anything, the mapv() function simply returns the reference of the source node_local itself in order to support method chaining while calling mapv().

Like map(), mapv() has exactly the same rules and limitations. It is only different in the sense that it accepts non-returning (void) function or function object. It can not be mapped on a function which returns something other than "void".

For example,

```
void func1(int x) { x = 2*x; // updates on temporary x local to func1() }
void func2(int& x) { x = 2*x; // in-place update }
int func3(int x) { return 2*x; }

// let's consider "nloc" is a node_local of integer type.
nloc.mapv(func1); // Ok, but "nloc" would remain unchanged.
nloc.mapv(func2); // Ok, all the worker data would get doubled.

// "nloc" is node_local<int>, func3() accepts int, but it also returns int.
// thus it can not be passed to a mapv() call.
nloc.mapv(func3); // error, func3() is a non-void function

// method chaining is allowed (since mapv returns reference to
// the source node_local)
auto r = nloc.mapv(func2).map(func3);
```

Here the resultant node_local "r" will be of integer type and it will contain 4 times the values stored in "nloc". While mapping func2() on the worker data of "nloc", it will get doubled in-place and the mapv() will return the reference of the updated "nloc" on which the map() function will apply the function func3() to double all the worker data once again (not in-place) and will return a new `node_local<int>`.

reduce()

It reduces all the worker data of the node_local object, by specifying some rule to be used for reduction. The rule can be any function or function object that satisfies associative law, like min, max, sum etc. with the below signatures.

```
T reduce(const F& f);
T reduce(T(*f)(TT,UU));
```

The type of the input/output of the input function defining the rule must be same or compatible with the type of the node_local object.

On success, it returns the reduced value of the same type of the node_local object.

For example,

```
int sum (int x, int y) { return x + y; }

std::vector<int> v_sum(const std::vector<int>& x,
                       const std::vector<int>& y) {
   std::vector<int> ret(x.size());
   for(auto i=0; i<x.size(); ++i) ret[i] = x[i] + y[i];
   return ret;
}

// let's consider "nloc1" is a node_local<int>
auto r1 = nloc1.reduce(sum);

// let's consider "nloc2" is a node_local<vector<int>>
auto r2 = nloc2.reduce(v_sum);
```

"r1" will be the reduced integer value of all the worker data as in "nloc1". Whereas "r2" will be the reduced integer vector of all the worker vector data as in "nloc2" as depicted below with two workers and with sample values (considering 5 is broadcasted to create "nloc1" and {{1,2},{3,4}} is scattered to create "nloc2"):

```
master                              worker0                 worker1
------                              -------                 -------
nloc1: node_local<int>              int: (5)                int: (5)
nloc2: node_local<vector<int>>      vector<int>: ({1,2})    vector<int>: ({3,4})
r1: int -> (10)
r2: vector<int> -> ({4,6})
```

Note, reducing a `dvector<int>` will result an integer value (e.g., 10 as in above case). Whereas, reducing a `node_local<vector<int>>` will result an integer vector (e.g., {4,6} as in above case) containing sum of each elements of the worker vector data.

vector_sum()

This is a short-cut function which can be used to reduce a `node_local<vector<T>>` using the associative rule of "sum". It can not be used on a node_local object of type other than `vector<T>`.

For example,

```
std::vector<int> v_sum(const std::vector<int>& x,
                       const std::vector<int>& y) {
   std::vector<int> ret(x.size());
   for(auto i=0; i<x.size(); ++i) ret[i] = x[i] + y[i];
   return ret;
}

// let's consider "nloc1" is a node_local<int> and
// "nloc2" is a node_local<vector<int>>
auto l1 = nloc1.vector_sum(); // error
auto l2 = nloc2.vector_sum(); // Ok
auto l3 = nloc2.reduce(v_sum); // Ok, same as "l2"
```

allreduce()

allreduce() can be considered as reducing the worker data of a node_local object and then broadcasting the reduced data to all the worker nodes to create a new node_local object.

Like reduce(), it also aims to reduce worker data with a reduction function or function object satisfying associative law, like min, max, sum etc. The reduction happens locally in this case. It has the below signture:

```
node_local<T> reduce(const F& f);
node_local<T> reduce(T(*f)(TT,UU));
```

The type of the input/output of the input function defining the rule must be same or compatible with the type of the node_local object.

On success, it returns a node_local object of the same type as in the source node_localobject, containing the reduced values at each worker nodes.

For example,

```
int sum (int x, int y) { return x + y; }

std::vector<int> v_sum(const std::vector<int>& x,
                       const std::vector<int>& y) {
   std::vector<int> ret(x.size());
   for(auto i=0; i<x.size(); ++i) ret[i] = x[i] + y[i];
   return ret;
}

// let's consider "nloc1" is a node_local<int>
auto r1 = nloc1.allreduce(sum);

// let's consider "nloc2" is a node_local<vector<int>>
auto r2 = nloc2.allreduce(v_sum);
```

"r1" will be a `node_local<int>` object containing the reduced values at each worker node for source node_local object "nloc1".

Whereas "r2" will be a `node_local<vector<int>>` object containing the reduced vectors at each worker node for the source node_local object "nloc2", as depicted below with two workers and with sample values (considering 5 is broadcasted to create "nloc1" and {{1,2},{3,4}} is scattered to create "nloc2"):

```
master                              worker0                 worker1
------                              -------                 -------
nloc1: node_local<int>              int: (5)                int: (5)
nloc2: node_local<vector<int>>      vector<int>: ({1,2})    vector<int>: ({3,4})
r1: node_local<int>                 int: (10)               int: (10)
r2: node_local<vector<int>>         vector<int>: ({4,6})    vector<int>: ({4,6})
```

Note that "broadcast(nloc2.reduce(v_sum))" is same as "nloc2.allreduce(v_sum)". But allreduce() attempts to reduce the elements of the worker data locally, thus it is more efficient and faster.

gather()

In order to gather the worker data of a node_local object one-by-one to the master node, gather() function can be used. It returns an std::vector of type T, where "T" is the type of the node_local object.

```
std::vector<T> gather();
```

For example,

```
// let's consider "nloc1" is a node_local<int>
auto r1 = nloc1.gather();

// let's consider "nloc2" is a node_local<vector<int>>
auto r2 = nloc2.gather();
```

"r1" will be a `vector<int>` containing the gathered integers from "nloc1" Whereas "r2" will be a `vector<vector<int>>` containing the gahered integer vectors from "nloc2" as depicted below with two workers and with sample values (considering 5 is broadcasted to create "nloc1" and {{1,2},{3,4}} is scattered to create "nloc2"):

```
master                              worker0                 worker1
------                              -------                 -------
nloc1: node_local<int>              int: (5)                int: (5)
nloc2: node_local<vector<int>>      vector<int>: ({1,2})    vector<int>: ({3,4})
r1: vector<int> -> ({5,5})
r2: vector<vector<int>> -> ({{1,2}, {3,4}})
```

Note, gathering a `dvector<int>` will result a `vector<int>` (e.g., {1,2,3,4} as in above case). Whereas, gathering a `node_local<vector<int>>` will result a `vector<vector<int>>` (e.g., {{1,2},{3,4}} as in above case) containing the vector chunk of each worker scattered data.

put()

This function can be used to modify or replace any existing worker data of a node_local object at a given position. It accepts the worker node id (zero-based) of the type "int" and the intended data to be inserted at that worker node for the source node_local object. It has the below signature:

```
void put(int nid, const T& val);
```

It allows user to perform a simple assignment like operation "nloc[nid] = val", where "nloc" is a node_local object. But such an operation should not be performed within a loop in order to avoid poor loop performance.

Here "nid" is the worker node id associated with the source node_local object. It's value must be within 0 to nproc-1, where "nproc" is the total number of participating nodes which can be obtained from "frovedis::get_nodesize()" call.

And "val" must be of the same or compatible type with the source node_local.

For example, if "nloc" is a `node_local<int>` created by broadcasting "5" among two worker nodes, then

```
// error, "nid" must be within 0 to nproc-1
nloc.put(frovedis::get_nodesize(),4);
nloc.put(0,2); // this will modify the node_local object as shown below

master                              worker0              worker1
------                              -------              -------
nloc: node_local<int>               int: (5)             int: (5)
(modified) nloc: node_local<int>    int: (2)             int: (5)
```

get()

This function can be used to get an existing worker data from a requested worker node associated with a node_local object. It has the below signature:

```
T get(int nid);
```

It is equivalent to an indexing operation "nloc[nid]", performed on a node_local object, "nloc". But such an operation should not be used within a loop in order to avoid poor loop performance.

Here "nid" is the target node id (0 to nproc-1) from which the node data is to be obtained. On success, it returns the data of the given position.

For example, if "nloc" is a `node_local<int>` created from broadcasting "5" among two worker nodes, then

```
auto r = nloc.get(1); // "r" will contain the 2nd worker data, "5"
auto x = nloc.get(2); // error, "nid" value must be within 0 to 1
```

as_dvector()

This function can be used to convert a `node_local<vector<T>>` to a `dvector<U>`, where type T and U must be same or compatible type. In this case, while converting to the dvector (see manual entry for dvector) object it copies the entire elements of the source `node_local<vector<T>>`. Thus after the conversion, source node_local will remain unchanged.

Note that, dvector conversion is possible only when the source node_local has vector chunk at associated worker nodes. And the type of the output dvector (U) has to be explicitly mentioned. The signature of the function is as follows:

```
dvector<U> as_dvector() const;
```

Let's consider "l1" is a `node_local<int>` and "l2" is a `node_local<vector<int>>`. Then,

```
auto dv1 = l1.as_dvector<int>(); // error
auto dv2 = l2.as_dvector<int>(); // Okay
```

Now let's consider "nloc" is a `node_local<vector<int>>` created from scattering {{1,2},{3,4}} among two worker nodes, then

```
void two_times_in_place(int& x) { x = 2*x; }

auto dv = nloc.as_dvector<int>(); // conversion to dvector<int> -> copy
// converted dvector elements will get doubled,
// but source node_local worker data will remain unchanged
dv.mapv(two_times_in_place);
```

```
master                              worker0                 worker1
-----                               -----                   -----
nloc: node_local<vector<int>>       vector<int>: ({1,2})    vector<int>: ({3,4})
(converted) dv: dvector<int>        vector<int>: ({1,2})    vector<int>: ({3,4})
(doubled)   dv: dvector<int>        vector<int>: ({2,4})    vector<int>: ({6,8})
```

moveto_dvector()

This function can be used to convert a `node_local<vector<T>>` to a `dvector<U>`, where type T and U must be same or compatible type. In this case, while converting to the dvector object it avoids copying the data in the source node_local. Thus after the conversion, source node_local object will become invalid. This is useful and faster when input node_local object will no longer be needed in a user program.

Note that, Like as_dvector() in this case also, dvector conversion is possible only when the source node_local has vector chunk at associated worker nodes. And the type of the output dvector (U) has to be explicitly mentioned. The signature of the function is as follows:

```
dvector<U> moveto_dvector();
```

Let's consider "l1" is a `node_local<int>` and "l2" is a `node_local<vector<int>>`. Then,

```
auto dv1 = l1.moveto_dvector<int>(); // error
auto dv2 = l2.moveto_dvector<int>(); // Okay
```

Now let's consider "nloc" is a `node_local<vector<int>>` created from scattering {{1,2},{3,4}} among two worker nodes, then

```
void two_times_in_place(int& x) { x = 2*x; }

auto dv = nloc.moveto_dvector<int>(); // conversion to dvector<int> -> move
// converted dvector elements will get doubled,
dv.mapv(two_times_in_place);
// but source node_local will become invalid
auto temp = nloc.gather(); // error (node_local data is moved, thus invalid)
```

```
master                              worker0                 worker1
------                              -------                 -------
nloc: node_local<vector<int>>       vector<int>: ({1,2})    vector<int>: ({3,4})
(converted) dv: dvector<int>        vector<int>: ({1,2})    vector<int>: ({3,4})
nloc: node_local<vector<int>>          ---                     ---
(doubled)   dv: dvector<int>        vector<int>: ({2,4})    vector<int>: ({6,8})
```

viewas_dvector()

This function can be used to create a view of a `node_local<vector<T>>` as a `dvector<U>`, where T and U must be of same or compatible type. Since it is about just creation of a view, the data in source node_local is neither copied nor moved. Thus it will remain unchanged after the view creation and any changes made in the source node_local will be reflected in its dvector view as well and the reverse is also true.

Note that, Like as_dvector() in this case also, dvector conversion is possible only when the source node_local has vector chunk at associated worker nodes. And the type of the output dvector (U) has to be explicitly mentioned. The signature of the function is as follows:

```
dvector<U> viewas_dvector();
```

Let's consider "l1" is a `node_local<int>` and "l2" is a `node_local<vector<int>>`. Then,

```
auto dv1 = l1.moveto_dvector<int>(); // error
auto dv2 = l2.moveto_dvector<int>(); // Okay
```

Now let's consider "nloc" is a `node_local<vector<int>>` created from scattering {{1,2},{3,4}} among two worker nodes, then

```
void two_times_in_place(int& x) { x = 2*x; }

void display_local(const std::vector<int>& v) {
  for (auto& e: v) std::cout << e << " ";
  std::cout << std::endl;
}

void display_global(int x) {
  std::cout << x << " ";
}

nloc.mapv(display_local); // node_local elements will be printed as 1 2 3 4
auto dv = nloc.viewas_dvector<int>(); // creation of a dvector<int> view
// "dv" and "nloc" both are refering to the same worker memory
// thus any changes in view "dv" will also be reflected in source "nloc"
dv.mapv(two_times_in_place);
dv.mapv(display_global); // dvector elements will be printed as 2 4 6 8
nloc.mapv(display_local); // node_local elements will be printed as 2 4 6 8
```

There might be a situation when some user function expects to receieve `dvector<T>` data just for reading, but input data is in `node_local<vector<T>>` form. In that case, this function will be useful just to create a dvector view and send to that user function for reading.

Public Global Function Documentation

node_local<T> make_node_local_allocate()

Purpose
This function is used to allocate empty T type instances at the worker nodes to create a valid empty node_local<T> at master node.

The default constructor of node_local, does not allocate any memory at the worker nodes. Whereas, this function can be used to create a valid empty node_local with allocated memory at worker nodes.

Note that, the intended type of the node_local object needs to be explicitly specified while calling this function.

For example,

```
void asign_data(std::vector<int>& v) {
  // get_selfid() returns rank of the worker node
  // which will execute this function
  auto myrank = frovedis::get_selfid(); // (0 to nproc-1)
  std::vector<int> temp;
  for(int i=1; i<=2; ++i) temp.push_back(i*myrank);
  v.swap(temp);
}

void display(const std::vector<int>& v) {
  for (auto& e: v) std::cout << e << " ";
  std::cout << std::endl;
}

node_local<vector<int>> nc1; // empty node_local without any allocated memory
// empty node_local with allocated memory
auto nc2 = make_node_local_allocate<vector<int>>();
nc1.mapv(display); // error, can't display "nc1" (it is not valid).
nc2.mapv(display); // okay, an empty view
// asigining data at each allocated empty partition and display contents
// if there are two worker nodes, it will display -> 0 0 1 2
nc2.mapv(asign_data).mapv(display);
```

Return Value
On success, it returns the allocated node_local<T>.

node_local<T> make_node_local_scatter(vec)

Parameters
vec: An std::vector<T> containing the elements to be scattered.

Purpose
This function accepts a normal vector of elements of type T and scatter them one-by-one to each participating worker node to create a node_local<T>. The size of the input vector must be same with the number of participating worker nodes, else an exception will be thrown. After the scattering, The input vector will remain unchanged.

Note that, the block size of each worker partition is auto decided by the frovedis when scattering a vector<T> to create a dvector<T>. But when a node_local object is to be created by scattering a vector data, user needs

to specify the same in chunk-per-worker, thus in that case the input argument has to be a `vector<vector<T>>` (instead of `vector<T>`).

For example, if there are two worker nodes, then

```
std::vector<int> v1 = {2,4};
auto nc1 = make_node_local_scatter(v1);// nc1 will be a node_local<int>
std::vector<std::vector<int>> v2 = {{1,2},{3,4}};
auto nc2 = make_node_local_scatter(v2);// nc2 will be a node_local<vector<int>>
std::vector<int> v3 = {2,4,6};
auto nc3 = make_node_local_scatter(v3);// error, vector size != worker size
```

```
master                            worker0               worker1
-----                             -----                 -----
v1: vector<int> ({2,4})
v2: vector<vector<int>> ({1,2},{3,4})
nc1: node_local<int>              int: (2)              int: (4)
nc2: node_local<vector<int>>      vector<int>: ({1,2})  vector<int>: ({3,4})
```

Return Value
On success, it returns the created `node_local<T>`.

`node_local<T>` make_node_local_broadcast(data)

Parameters
data: A const& of a "T" type data to be broadcasted.

Purpose
This function accepts a T type data and broadcasts it to each participating worker node to create a `node_local<T>`.

For example, if there are two worker nodes, then

```
std::vector<int> v = {1,2};
auto nc1 = make_node_local_broadcast(2);// nc1 will be a node_local<int>
auto nc2 = make_node_local_broadcast(v);// nc2 will be a node_local<vector<int>>
```

```
master                            worker0               worker1
-----                             -----                 -----
v: vector<int> ({1,2})
nc1: node_local<int>              int: (2)              int: (2)
nc2: node_local<vector<int>>      vector<int>: ({1,2})  vector<int>: ({1,2})
```

Note that, there is a short-cut method, called "broadcast()" to perform the same thing. For example, make_node_local_broadcast(t) and broadcast(t) both are equivalent.

Return Value
On success, it returns the created `node_local<T>`.

SEE ALSO

dvector, dunordered_map