frovedis::dvector<T>

NAME

`frovedis::dvector<T>` - a distributed vector of type 'T' supported by frovedis

SYNOPSIS

`#include <frovedis.hpp>`

Constructors

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

Overloaded Operators

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

Public Member Functions

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

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

`template <class R, class U, class V, class F>`
`dvector<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>`
`dvector<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>`
`dvector<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>`
`dvector<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>
dvector<R> map(R(*f)(TT));

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

template <class R, class U, class V, class TT, class UU, class VV>
dvector<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>
dvector<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>
dvector<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>
dvector<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>
dvector<T>& mapv(const F& f);

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

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

template <class U, class V, class W, class F>
dvector<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>
dvector<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>
dvector<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>
dvector<T>& mapv(void(*f)(TT));

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

template <class U, class V, class TT, class UU, class VV>
dvector<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>
dvector<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>
dvector<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>
dvector<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 R, class F>
dvector<R> map_partitions(const F& f);

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

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

template <class R, class U, class V, class W, class F>
dvector<R> map_partitions(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>
dvector<R> map_partitions(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>
dvector<R> map_partitions(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>
dvector<R> map_partitions(std::vector<R>(*f)(TT));

template <class R, class U, class TT, class UU>
dvector<R> map_partitions(std::vector<R>(*f)(TT, UU),
        const node_local<U>& l);

template <class R, class U, class V, class TT, class UU, class VV>
dvector<R> map_partitions(std::vector<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>
dvector<R> map_partitions(std::vector<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>
dvector<R> map_partitions(std::vector<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>
dvector<R> map_partitions(std::vector<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>
dvector<T>& mapv_partitions(const F& f);

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

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

template <class U, class V, class W, class F>
dvector<T>& mapv_partitions(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>
dvector<T>& mapv_partitions(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>
dvector<T>& mapv_partitions(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>
dvector<T>& mapv_partitions(void(*f)(TT));

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

template <class U, class V, class TT, class UU, class VV>
dvector<T>& mapv_partitions(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>
dvector<T>& mapv_partitions(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>
dvector<T>& mapv_partitions(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>
dvector<T>& mapv_partitions(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> dvector<T> filter(const F& f);
template <class TT> dvector<T> filter(bool(*f)(TT));

template <class F> dvector<T>& inplace_filter(const F& f);
template <class TT> dvector<T>& inplace_filter(bool(*f)(TT));

template <class R, class F> dvector<R> flat_map(const F& f);
template <class R, class TT> dvector<R> flat_map(std::vector<R>(*f)(TT));

void clear();
std::vector<T> gather();
const std::vector<size_t>& sizes();
size_t size() const;
dvector<T>& align_as(const std::vector<size_t>& sz);
template <class U> dvector<T>& align_to(dvector<U>& target);
dvector<T>& align_block();

void save(const std::string& path, const std::string& delim);
void saveline(const std::string& path);
void savebinary(const std::string& path);
void put(size_t pos, const T& val);
T get(size_t pos);

template <class K,class V>
dunordered_map<K,std::vector<V>> group_by_key();
template <class K,class V,class F>
dunordered_map<K,V> reduce_by_key(const F& f);
template <class K,class V,class VV,class WW>
dunordered_map reduce_by_key(V(*f)(VV,WW));

node_local<std::vector<T>> as_node_local() const;
node_local<std::vector<T>> moveto_node_local();
node_local<std::vector<T>> viewas_node_local();

dvector<T>& sort(double rate = 1.1);
template <class F> dvector<T>& sort(F f, double rate = 1.1);
```

DESCRIPTION

`frovedis::dvector<T>` can be considered as the distributed version of `std::vector<T>`. Memory management is similar to vector (RAII): when a dvector is destructed, the related distributed data is deleted at the time. It is possible to copy or construct it from an existing dvector. In this case, distributed data is also copied (if the source variable is an rvalue, the system tries to avoid copy).

The dvector can also be created while loading data from file. When a vector of size 4, e.g., {1,2,3,4} is distributed among two worker nodes, worker0 will have {1,2} and worker1 will have {3, 4}. Frovedis supports various member functions of a dvector to develop interesting programs.

The dvector provides a global view of the distributed vector to the user. When operating on a dvector, user can simply specify the intended operation on each element of the dvector (not on each local partition of the worker data). Thus it is simpler to handle a dvector like an std::vector, even though it is distributed among multiple workers. Every dvector has a “size vector” attribute, containing the size of each local vectors at worker nodes. The next section explains its functionalities in details.

Constructor Documentation

dvector ()

This is the default constructor which creates an empty dvector. But it does not allocate data, like normal container. See make_dvector_allocate().

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

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

dvector (`dvector<T>`&& src)

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

Overloaded Operator Documentation

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

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

`dvector<T>`& operator= (`dvector<T>`&& src)

Instead of copying, it moves the contents of the source rvalue dvector object into the left-hand side target dvector object of the assignment operator "=". It is faster and recommended when source dvector object will no longer be needed. It returns the reference of the target dvector 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 element of a dvector. It accepts a function or a function object (functor) and applies the same to each element of the dvector in parallel at the workers. Then a new dvector 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.

```
dvector<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 argument.
- the type of the function argument must be same or compatible with the type of the dvector.
- the return type, R can be anything. The resultant dvector 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 "dv" is a dvector of type float
// dv is dvector<float>, func1() accepts float.
auto r1 = dv.map(func1); // Ok, r1 would be dvector<float>.

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

// dv is dvector<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 = dv.map(func3); // error

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

// it is possible to chain the map calls
auto r5 = dv.map(func1).map(func4); // Ok, r5 would be dvector<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.

```
dvector<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 dvector.
- the type of the dvector 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 dvector 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 dvector, "dv" will be as follows:

```
 func(d,x1,x2,x3,x4,x5);      dv.map(func,l1,l2,l3,l4,l5);
 ----------------------       ----------------------
    d: T                         dv: dvector<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,

```
int func1(int x, int y) { return x+y; }
double func2(int x, float y, double z) { return x*y+z; }

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

// broadcasting integer "y" to all workers to obtain node_local<int>.
int y = 2;
auto dy = broadcast(y);
auto r2 = dv.map(func1, dy); // Ok, r2 would be dvector<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 = dv.map(func2, dy, dz); // Ok, r3 would be dvector<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.

  dvector map(const F& f);

- the type of the dvector 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.

  dvector 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 dvector 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) {}
  double operator() (int x) { // 1st definition
    return al*x+be;
  }
  double operator() (int x, int y) { // 2nd definition
    return al*x+be*y;
  }
  float al, be;
  SERIALIZE(al,be)
};

// let's consider "dv" is a dvector of "int" type.
auto r1 = dv.map<double>(foo(2.0,3.0)); // ok, r1 would be dvector<double>
```

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.

"dv" is `dvector<int>` and map() is called with only functor definition. Thus it will hit the first definition of the overloaded "operator()". The return type is "double" which can be of any type and needs to be explicitly mentioned while calling the map() function like `map<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
// dvector<double>.
auto r2 = dv.map<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.

mapv()

The mapv() function is also used to specify the target operation to be mapped on each element of the dvector. It accepts a void returning function or a function object (functor) and applies the same to each element of the dvector in parallel at the workers. Since the applied function does not return anything, the mapv() function simply returns the reference of the source dvector 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 "dv" is a dvector of integer type.
dv.mapv(func1); // Ok, but "dv" would remain unchanged.
dv.mapv(func2); // Ok, all the elements of "dv" would be doubled.

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

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

Here the resultant dvector “r” will be of integer type and it will contain 4 times the values stored in “dv”. While mapping func2() on the elements of “dv”, it will double all its elements in-place and the mapv() will return the reference of the updated “dv” on which the map() function will apply the function func3() to double all its elements once again (not in-place) and will return a new `dvector<int>`.

map_partitions()

Unlike map() function, map_partitions() accept the function or function object to be mapped on each partition of the dvector (not on each element of the dvector). Thus the input function (or functor) must accept an std::vector of type T (as the first argument) and must return an std::vector of type R. Where “T” must be the same or compatible with the type of the dvector and “R” can be of any type, the resultant dvector would be of same type.

For example,

```
std::vector<double> func(std::vector<int> part) {
   std::vector<double> ret(part.size());
   for(size_t i=0; i<part.size(); ++i) ret[i] = 2.0 * part[i];
   return ret;
}

// let’s consider "dv" is a dvector of type integer.
// mapping the rule defined by func() in each partition of dv.
auto r = dv.map_partitions(func); // "r" would be of type dvector<double>
```

Like map() function, it also has similar rules and limitations.

- The first argument of the map_partitions() must be a function or function object to be mapped on each partition of the dvector.
- Apart from the function (or function object), it can accept a maximum of five distributed node_local objects of any type which must be same or compatible with the corresponding types of the input function arguments.

- In case more than five arguments are required to be passed, a function object can be defined by setting all the required values and can be passed to the map_partitions() call, as explained in earlier in the map section.
- Usually function object version is a bit faster, but special treatment like explicit specification of the return type etc. (as explained in map section) needs to be considered.

Usually map_partitions() can work faster than map() on SX, since in case of map_partitions() user needs to pass a function defining rules to be mapped on each element of a specific partition. The do-loop (to iterate over a partition) inside such functions can be vectorized, whereas map() allows user to simply define the rule to be mapped on each element of the dvector without any do-loop to iterate over the partition.

mapv_partitions()

Like map_partitions(), it also accepts the function or function object to be mapped on each partition of the dvector (not on each element of the dvector). But in this case, the input function (or functor) must accept an std::vector of type T (as the first argument) where "T" must be the same or compatible with the type of the dvector and it must not return anything (void returning function).

Like map_partitions(), mapv_partitions() has exactly the same rules and limitations. It is only different in the sense that it accepts non-returning (void) function or function object. Although it can not be mapped on a function which returns something other than "void", the mapv_partitions() itself returns the reference of the source dvector in order to support method chaining.

For example,

```
void func1(std::vector<int> part) {
   // update on a temporary "part" local to the func1()
   for(size_t i=0; i<part.size(); ++i) part[i] *= 2;
}

void func2(std::vector<int>& part) {
   // in-place update on the "part" itself
   for(size_t i=0; i<part.size(); ++i) part[i] *= 2;
}

std::vector<double> func3(std::vector<int> part) {
   std::vector<double> ret(part.size());
   for(size_t i=0; i<part.size(); ++i) ret[i] = 2.0 * part[i];
   return ret;
}

// let's consider "dv" is a dvector of type integer.
// mapping the rule defined by func1() in each partition of dv.
dv.mapv_partitions(func1); // Ok, but "dv" will remain unchanged

// mapping the rule defined by func2() in each partition of dv.
dv.mapv_partitions(func2); // Ok, "dv" will get doubled in-place

// mapping the rule defined by func3() in each partition of dv.
dv.mapv_partitions(func3); // error, func3() is not a void function

// since mapv_partitions() returns the reference of the source
// dvector itself, the method chaining is possible.
auto r = dv.mapv_partitions(func2).map_partitions(func3);
```

Here the resultant dvector "r" will be of double type and it will contain 4 times the values stored in "dv". While mapping func2() on the elements of "dv", it will double all its partitions in-place and the mapv_partitions() will return the reference of the updated "dv" on which the map_partitions() function will apply the function func3() to double all its partitions once again (not in-place) and will return a new dvector.

flat_map()

Like map(), flat_map() can also be used to map a user function on each elements of a dvector. But in this case, the user function must return more than one values in a vector form while mapping the function against each elements. It flattens the output vector returned by the user function while constructing the resultant dvector. For this reason, the size of the resultant vector is larger than the size of the source dvector.

Unlike map(), flat_map() can accept only the user function or function object to be mapped on each dvector elements. The type of the argument of the user function and the type of the dvector must be same or compatible. The return type, "R" can be anything. The resultant dvector will be of the same type.

```
dvector<R> flat_map(const F& f);
dvector<R> flat_map(std::vector<R>(*f)(TT));
```

For example,

```
// function (returning a vector) to be mapped on a dvector
std::vector<int> duplicate (int i) {
  std::vector<int> ret;
  ret.push_back(i); ret.push_back(2*i);
  return ret;
}

// let's consider "dv" is a dvector of type "int".
auto r1 = dv.map(duplicate); // r1 will be dvector<std::vector<int>>
auto r2 = dv.flat_map(duplicate); // r2 will be dvector<int>
```

Let's consider a vector of integers {1,2,3,4} is scattered over two workers to create the dvector "dv". Then worker0 will have {1,2} and worker1 will have {3,4}. Now in case of the resultant dvector, "r1" from "map(duplicate)" opetration, worker0 will have {{1,2},{2,4}} and worker1 will have {{3,6},{4,8}} and it would be of the type std::vector and both "dv" and "r1" will have the same size (4 in this case). But in case of the resultant dvector, "r2" from flat_map(duplicate) operation, worker0 will have {1,2,2,4} and worker1 will have {3,6,4,8} and it would be of the type "int" with double the size of "dv".

reduce()

It reduces all the elements in the dvector to a single scalar value, 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 dvector.

On success, it returns the reduced scalar value of the same type of the dvector.

For example,

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

// let’s consider "dv" is a dvector of type "int"
// "r" would be an integer value containing the summed-up of all the elements
// in the dvector, "dv".
auto r = dv.reduce(sum);
```

filter()

Some specific elements from a dvector can be filtered out with the help of filter() function. It accepts a function or a function object specifying the condition on which the element is to be filtered out from the dvector. The type of the function argument must be same or compatible with the type of the dvector and the function must return a boolean value (true/false).

```
dvector<T> filter(const F& f);
dvector<T> filter(bool(*f)(TT));
```

On success, it will return a new dvector of same type containing the filtered out elements.

For example,

```
bool is_even(int n) { return n%2 == 0; }

// let’s consider "dv" is a dvector of type "int"
// r will be the resultant dvector<int> containing only the even numbers
// from the target dvector "dv".
auto r = dv.filter(is_even);
```

inplace_filter()

Like filter(), this function can also be used to filter out some specific elements from a dvector. But in this case the filtration happens in-place, i.e., instead of returning a new dvector, this function aims to update the source dvector by keeping only the filtered out elements in it.

Like filter(), it also accepts a function or a function object specifying the condition on which the element is to be filtered out from the dvector. The type of the function argument must be same or compatible with the type of the dvector and the function must return a boolean value (true/false).

```
dvector<T>& inplace_filter(const F& f);
dvector<T>& inplace_filter(bool(*f)(TT));
```

On success, the source dvector will be updated with only the filtered out elements in-place and this function will return a reference to itself.

For example,

```
bool is_even(int n) { return n%2 == 0; }

// let’s consider "dv" is a dvector of type "int" containing both even
// and odd numbers. it will contain only the even numbers after the below
// in-place operation.
dv.inplace_filter(is_even);
```

clear()

In order to remove the existing elements and clear the memory space occupied by a dvector, clear() function can be used. It returns void.

gather()

In order to gather the distributed vector data from all the workers to master process, gather() function can be used. It returns an std::vector of type T, where “T” is the type of the dvector.

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

Data gathering happens worker-by-worker. For example if there are two worker nodes and worker0 has {1,2} and worker2 has {3,4}, then performing gather() on that dvector will result in a vector containing {1,2,3,4}.

sizes()

This function returns the size vector of an existing dvector containing the size of each local vectors at worker nodes. It has the below signature:

```
const std::vector<size_t>& sizes();
```

For example if “dv” is a `dvector<int>` distributed over two worker nodes and worker0 has {1,2,3,4} and worker1 has {5,6}, then calling sizes() on “dv” will result in a a vector containing {4,2}, i.e., the sizes of each local vectors. The size of the output vector will be same as the number of participating worker nodes. For example if the above “dv” is distributed over four worker nodes and worker0 has {1,2,3,4}, worker1 has {5,6} and worker2 and worker3 doesn’t contain any part of this distributed vector, then calling size() on that dvector will result in a vector containing {4,2,0,0}.

size()

This function returns the size of the distributed vector. The signature of this function is as follows:

```
size_t size() const;
```

For example, if `std::vector<int>` {1,1,0,1} is distributed among workers to create a `dvector<int>` “dv”, then dv.size() will return 4.

align_as()

This function can be used to re-align the distribution of an existing dvector. It accepts an `std::vector<size_t>` containing the sizes of the local vectors as per the desired re-alignment.

The function will work well, only when below conditions are true:

- the size of the input vector must match with the number of worker nodes.
- the size of the source dvector must also match with the size of the desired re-aligned dvector.

On success, it will return a reference to the re-aligned dvector. The signature of the function is as follows:

```
dvector<T>& align_as(const std::vector<size_t>& sz);
```

For example, if `std::vector<int>` {1,1,0,1} is distributed among 2 worker nodes to create a `dvector<int>` “dv”, then

```
auto r1 = dv.sizes(); // it will return size vector of "dv" -> {2,2}
std::vector<size_t> t1 = {3,1};
auto r2 = dv.align_as(t1).sizes(); // Ok, it will return {3,1}
std::vector<size_t> t2 = {2,1,1};
// it will throw an exception, since size of t2 is 3,
// but number of workers is 2
r2 = dv.align_as(t2).sizes(); // error
std::vector<size_t> t3 = {3,2};
// it will throw an exception, since size of input dvector was 4,
// but provided size after re-alignment is 3+2 = 5
r2 = dv.align_as(t3).sizes(); // error
```

align_to()

This function is used to re-align an existing dvector, “v1” according to the
alignment of another existing dvector, “v2”. The type of “v1” and “v2” can differ, but their size must be same in order to perform a re-alignment.

On success, it will return the reference to the re-aligned dvector “v1”. The signature of this function is as follows:

```
template <class U> dvector<T>& align_to(dvector<U>& target);
```

For example, if size vector of “v1” is {2,2} and the size vector of “v2” is {3,1}, then

```
v1.align_to(v2);
// "v1" will get re-aligned according to "v2"
auto r = v1.sizes(); // it will return {3,1}
```

align_block()

This function is used to re-align an existing dvector according to the frovedis default alignment.

If the target dvector is of the size 10 and the number of worker nodes is 4, then frovedis computes the chunk size per worker node according to the formula “ceil(size_of_dvector/num_of_worker)”, which would be evaluated as 3 in this case [ceil(10/4)].

So worker0 will contain the first 3 elements, worker1 will contain next 3 elements, worker2 will contain next 3 elements and worker3 will contain the remaining last element. Therefore, the size vector after this re-alignment will be {3,3,3,1}.

On success, it will return the reference to the re-aligned dvector. If the size vector of the target dvector already is in frovedis default alignment, then no operation will be performed. Simply the reference to the target dvector would be returned. The signature of this function is as follows:

```
dvector<T>& align_block();
```

For example, let’s consider “dv” is a `dvector<int>` of size 4 distributed among four worker nodes. Then,

```
std::vector<size_t> tmp = {2,2,0,0};
auto r1 = dv.align_as(tmp).sizes(); // it will return {2,2,0,0}
auto r2 = dv.align_block().sizes(); // it will return {1,1,1,1}
```

save()

In order to dump the contents of a dvector in a file in readable text form, this function can be used. The signature of this funtion is as follows:

```
void save(const std::string& path,
          const std::string& delim);
```

It accepts the absolute/relative path of the filename where to write the contents, along with the delimeter by which two consecutive elements of the dvector is to be separated while writing into the specified file.

For example, if "dv" is a `dvector<int>` created from a vector {1,2,3,4}, then dv.save("./sample","\n"); will write the content of the dvector in the specified file "sample" with a new line character after each element as follows:

```
1
2
3
4
```

saveline()

saveline() is a short-cut version of the function save(). While in case of save(), the delimeter value is required to be provided, saveline() writes the contents of the dvector with a new line character after each element.

The signature of this function is as follows:

```
void saveline(const std::string& path);
```

It only accepts the absolute/relative path of the file where to write the contents of the dvector. For example, saveline("./sample") is same as save("./sample","\n").

savebinary()

Unlike save(), savebinary() writes the contents of the dvector in the specified file in non-readable binary (little-endian) form.

The signature of the function is as follows:

```
void savebinary(const std::string& path);
```

It only accepts the absolute/relative path of the file where to write the contents of the dvector in binary form.

put()

This function can be used to modify or replace any existing element of the dvector at a given position. It accepts the position (zero-based) of the type "size_t" and the element to be inserted at that position. It has the below signature:

```
void put(size_t pos, const T& val);
```

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

Here "pos" is the position where the element is to be put. It's value must be within 0 to size-1 of the dvector. And "val" must be of the same or compatible type with the dvector.

For example, if "dv" is a `dvector<int>` created from {1,2,3,4}, then

```
dv.put(2,2); // this will modify the dvector as -> {1,2,2,4}
dv.put(4,4); // error, "pos" value must be within 0 to 3
```

get()

This function can be used to get an existing element of the dvector from a given position (zero-based). It has the below signature:

```
T get(size_t pos);
```

It is equivalent to an indexing operation "dv[pos]", performed on a distributed vector, "dv". But such an operation should not be used within a looe in order to avoid poor loop performance.

Here "pos" is the position (0 to size-1) from which the element is to be obtained. On success, it returns the element of the given position.

For example, if "dv" is a `dvector<int>` created from {1,2,3,4}, then

```
auto r = dv.get(2); // "r" will contain the 3rd element of the dvector, 3
auto x = dv.get(4); // error, "pos" value must be within 0 to 3
```

group_by_key()

When a vector containing key-value pairs with key-type K and value-type V is distributed among participating worker nodes to create a `dvector<std::pair<K,V>>`, it is possible to group the values based on the unique keys in that dvector using group_by_key() member function.

On success, the output will be a distributed unordered_map like structure containing the group of values corresponding to each unique key. In frovedis, such an unordered_map is represented as `dunordered_map<K,vector<V>>` (see manual of dunordered_map). The signature of the function is as follows:

```
dunordered_map<K,std::vector<V>> group_by_key();
```

For example,

```
std::vector<std::pair<int,int>> v;
v.push_back(make_pair(1,100));
v.push_back(make_pair(2,200));
v.push_back(make_pair(1,300));
v.push_back(make_pair(2,400));
auto m =  make_dvector_scatter(v).group_by_key<int,int>;
```

Here "m" is an object of `dunordered_map<int,vector<int>>` type with the below contents:

```
1: {100, 300}
2: {200, 400}
```

Note that, it will be required to explicitly specify the key-value types when calling the function, as shown in the example above `group_by_key<int,int>` (else some compilation error can be experienced).

There is also a non-member function (global function in frovedis namespace) which accepts the source dvector as input argument as follows:

```
dunordered_map<K,std::vector<V>> group_by_key(dvector<std::pair<K,V>& dv);
```

It can be called without explicitly specifying the key-value types of the resultant dunordered_map, as follows:

```
auto dv = make_dvector_scatter(v);
auto m2 = frovedis::group_by_key(dv); // no need for explit key-value type
```

reduce_by_key()

As explained above, the values corresponding to a unique key in a `dvector<std::pair<K,V>>` can be grouped together using group_by_key(). Similarly the values associated with a unique key can be reduced using reduce_by_key() function passing a user defined reduction function (or function object) satisfying the associative laws like sum, min, max etc.

On success, it will return an object of the type `dunordered_map<K,V>` containing the distributed unordered_map where every key of K type will have an associated reduced value of V type.

Note that, in case of group_by_key(), the key-value types of the resultant dunordered_map are in `<K,vector<V>>` form. Whereas, in case of reduce_by_key(), these are in `<K,V>` form since it reduces all the values in a group associated with a unique key. The signature of the function is as follows:

```
dunordered_map<K,V> reduce_by_key(const F& f);
dunordered_map<K,V> reduce_by_key(V(*f)(VV,WW));
```

The input/output type of the user specified reduction function must be same or compatible with the value-type in the target dvector containing key-value pairs. And the resultant dunordered_map key-value type must be explicitly specified when calling the reduce_by_key() function (else some compilation error might be experienced).

For example,

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

int vec_sum(int k, std::vector<int>& v) {
  int res = 0; for(auto& i: v) res += i; return res;
}

std::vector<std::pair<int,int>> v;
v.push_back(make_pair(1,100));
v.push_back(make_pair(2,200));
v.push_back(make_pair(1,300));
v.push_back(make_pair(2,400));
auto m =  make_dvector_scatter(v).reduce_by_key<int,int>(sum);
```

Here “m” is an object of `dunordered_map<int,int>` type with the below contents:

```
1: 400 -> reduced sum value of the group {100,300}
2: 600 -> reduced sum value of the group {200,400}
```

Calling reduce_by_key() is equivalent to the below operation:

```
auto m2 = make_dvector_scatter(v).group_by_key<int,int>()
                                 .map_values(vec_sum);
```

But from the implementation point of view, reduce_by_key() is better than calling group_by_key() and then reducing the vector, because the system tries to reduce the data locally at first, which reduces the communication cost.

Note that, it will be required to explicitly specify the key-value types when calling the function, as shown in the example above `reduce_by_key<int,int>`.

There is also a non-member function (global function in frovedis namespace) which accepts the source dvector and the reduction rule as functor (function object) as follows:

```
dunordered_map<K,V> reduce_by_key(dvector<std::pair<K,V>& dv,
                                  const F& f);
```

It can be called without explicitly specifying the key-value types of the resultant dunordered_map, as follows:

```
struct sumtor {
  sumtor() {}
  int operator() (intx, int y) { return x + y; }
  SERIALIZE_NONE
};
auto dv = make_dvector_scatter(v);
auto m3 = frovedis::reduce_by_key(dv,sumtor()); // no need for explit key-value type
```

as_node_local()

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

The signature of the function is as follows:

```
node_local<std::vector<T>> as_node_local() const;
```

For example, if "dv" is a `dvector<int>` created from {1,2,3,4}, then

```
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 << " ";
}

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

```
dv.mapv(display_global); // dvector elements will be printed as 1 2 3 4
auto nloc = dv.as_node_local(); // conversion to node_local -> copy
dv.mapv(two_times_in_place); // dvector elements will get doubled
dv.mapv(display_global); // dvector elements will be printed as 2 4 6 8

// node_local elements will be printed as in original dvector 1 2 3 4
nloc.mapv(display_local);
```

moveto_node_local()

This function can be used to convert a `dvector<T>` to a `node_local<std::vector<T>>`, where "T" can be of any type. In this case, while converting to the node_local (see manual entry for node_local) object it avoids copying the data in the source dvector. Thus after the conversion, source dvector will become invalid. This is useful and faster when input node_local object will no longer be needed in a user program.

The signature of the function is as follows:

```
node_local<std::vector<T>> moveto_node_local();
```

For example, if "dv" is a `dvector<int>` created from {1,2,3,4}, then

```
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 << " ";
}

void two_times_in_place(int& x) { x = 2*x; }
void two_times_in_place_part(std::vector<int>& v) {
  for(auto i=0; i<v.size(); ++i) v[i] *= 2;
}

dv.mapv(display_global); // dvector elements will be printed as 1 2 3 4
auto nloc = dv.moveto_node_local(); // conversion to node_local -> no copy
dv.mapv(two_times_in_place); // error, "dv" will no longer be a valid dvector

nloc.mapv(display_local); // node_local elements will be printed as 1 2 3 4
nloc.mapv(two_times_in_place_part); // node_local elements will get doubled
nloc.mapv(display_local); // node_local elements will be printed as 2 4 6 8
```

viewas_node_local()

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

The signature of the function is as follows:

```
node_local<std::vector<T>> viewas_node_local();
```

For example, if “dv” is a `dvector<int>` created from {1,2,3,4}, then

```
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 << " ";
}

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

dv.mapv(display_global); // dvector elements will be printed as 1 2 3 4
auto nloc = dv.viewas_node_local(); // will create a node_local view
// "dv" and "nloc" both are refering to the same worker memory
// thus any changes in view "nloc" will also be reflected in source "dv"
dv.mapv(two_times_in_place); // dvector elements (in view) will get doubled
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 `node_local<vector<T>>` data just for reading, but input data is in `dvector<T>` form. In that case, this function will be useful just to create a node_local view and send to that user function for reading.

sort()

This function can be used to sort the elements of the dvector. It has the below signature:

```
dvector<T>& sort(double rate = 1.1);
```

“rate” is a double type parameter used internally while sorting of the dvector.

Frovedis also supports another version of sort() which accepts a function object defining the user given sort function, as follows:

```
dvector<T>& sort(F f, double rate = 1.1);
```

On success, this will sort the dvector in-place and will return a reference to the sorted dvector. Note, currently sort() is not supported on a view.

Public Global Function Documentation

`dvector<T>` make_dvector_allocate()

Purpose
This function is used to allocate empty vector instances of type “T” at the worker nodes to create a valid empty `dvector<T>` at master node.

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

Note that, the intended type of the dvector 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(int e) { std::cout << e << " "; }

dvector<int> dv1; // empty dvector without any allocated memory
auto dv2 = make_dvector_allocate<int>(); // empty dvector with allocated memory
dv1.mapv(display); // error, can't display "dv1" (it is not valid).
dv2.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
dv2.mapv_partitions(asign_data).mapv(display);
```

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

`dvector<T>` make_dvector_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 to the participating worker nodes to create a `dvector<T>`. Before scattering the data it partitions the data in blocks, as explained in align_block() function above. The input vector will remain unchanged.

For example,

```
void display(const std::vector<int>& v) {
   auto myrank = frovedis::get_selfid();
   for (auto& e: v) std::cout << "[" << myrank << "]: " << e << "\n";
}

std::vector<int> v = {1,2,3,4};
auto dv = make_dvector_scatter(v);
dv.mapv_partitions(display);
```

If there are two worker nodes, it will output (order of the display can be different):

```
[0]: 1
[0]: 2
```

```
[1]: 3
[1]: 4

master                           worker0              worker1
-----                            -----                -----
v: vector<int> ({1,2,3,4})
dv: dvector<int>                 vector<int>: ({1,2})   vector<int>: ({3,4})
```

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

`dvector<T>` **make_dvector_loadline(path)**

Parameters
path: A string object containing the path of the file to be loaded

Purpose
This function accepts a filename (with relative/absolute path) and loads the data from the text file to create a `dvector<T>` object. It expects the vector elements in the given file is separated by new-lines. If "T" is not explicitly provided, then it creates a `dvector<std::string>` type object. Otherwise, it creates a `dvector<T>`.

For example, if the text file "sample" contains below data
1
2
3
4

Then,

```
auto dv1 = make_dvector_loadline("./sample"); // dv1: dvector<std::string>
auto dv2 = make_dvector_loadline<int>("./sample"); // dv2: dvector<int>
```

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

`dvector<T>` **make_dvector_load(path,delim)**

Parameters
path: A string object containing the path of the file to be loaded
delim: A string object containing the delimeter used in the file for vector elements

Purpose
This function accepts a filename (with relative/absolute path), along with a delimeter of string type and loads the data from the text file based on the delimeter to create a `dvector<T>` object. If "T" is not explicitly provided, then it creates a `dvector<std::string>` type object. Otherwise, it creates a `dvector<T>`.

For example, if the text file "sample" contains below data
1, 2, 3, 4

Then,

```
auto dv1 = make_dvector_load("./sample", ","); // dv1: dvector<std::string>
auto dv2 = make_dvector_load<int>("./sample", ","); // dv2: dvector<int>
```

Note than `make_dvector_loadline<T>(filename)`, internally calls `make_dvector_load<T>(filename,"\n")`.

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

SEE ALSO

node_local, dunordered_map