Tutorial on the MPL interface to MPI

Victor Eijkhout

PEARC 2023

While the C API to MPI is usable from C++, it feels very unidiomatic for that language. Message Passing Layer (MPL) is a modern C++11 interface to MPI. It is both idiomatic and elegant, simplifying many calling sequences.

Part I

Basics

For doing the exercises:

```
module load mpl
```

which defines `TACC_MPL_INC`, `TACC_MPL_DIR`

To compile MPL programs, add a line

```
1  #include <mpl/mpl.hpp>
```

to your file.

TACC

There is no initialization or finalize call.

MPL implementation note: Initialization is done at the first `mpl::environment` *method call, such as* `comm_world`*.*

The naive way of declaring a communicator would be:

```
1  // commrank.cxx
2  mpl::communicator comm_world =
3    mpl::environment::comm_world();
```

calling the predefined environment method `comm_world`.

However, if the variable will always correspond to the world communicator, it is better to make it `const` and declare it to be a reference:

```
1  const mpl::communicator &comm_world =
2    mpl::environment::comm_world();
```

Processor name

TACC

The `processor_name` call is an environment method returning a `std::string`:

```
std::string mpl::environment::processor_name ();
```

The rank of a process (by `mpl::communicator::rank`) and the size of a communicator (by `mpl::communicator::size`) are both methods of the `communicator` class:

```
const mpl::communicator &comm_world =
    mpl::environment::comm_world();
int procid = comm_world.rank();
int nprocs = comm_world.size();
```

```
int mpl::communicator::size ( ) const
```

TACC

```
int mpl::communicator::rank ( ) const
```

The timing routines `wtime` and `wtick` and `wtime_is_global` are environment methods:

```
1  double  mpl::environment::wtime ();
2  double  mpl::environment::wtick ();
3  bool mpl::environment::wtime_is_global ();
```

The `environment` namespace has the equivalents of `MPI_COMM_WORLD` and `MPI_COMM_SELF`:

```
1  const communicator& mpl::environment::comm_world();
2  const communicator& mpl::environment::comm_self();
```

There doesn't seem to be an equivalent of `MPI_COMM_NULL`.

TACC

The communicator class has its copy operator deleted; however, copy initialization exists:

```cxx
// commcompare.cxx
const mpl::communicator &comm =
  mpl::environment::comm_world();
cout << "same: " << boolalpha << (comm==comm) << endl;

mpl::communicator copy =
  mpl::environment::comm_world();
cout << "copy: " << boolalpha << (comm==copy) << endl;

mpl::communicator init = comm;
cout << "init: " << boolalpha << (init==comm) << endl;
```

(This outputs true/false/false respectively.)

MPL implementation note: The copy initializer performs an `MPI_Comm_dup`.

TACC

Communicators can be duplicated but only during initialization. Copy assignment has been deleted. Thus:

```
1   // LEGAL:
2   mpl::communicator init = comm;
3   // WRONG:
4   mpl::communicator init;
5   init = comm;
```

TACC

Pass communicators by reference to avoid communicator duplication:

```cpp
// commpass.cxx
// BAD! this does a MPI_Comm_dup.
void comm_val( const mpl::communicator comm );

// correct!
void comm_ref( const mpl::communicator &comm );
```

Part II

Collectives

Collectives have many polymorphic variants, for instance for 'in place', and buffer handling.

Operators are handled through functors.

Buffer type handling is done through polymorphism and templating: no explicit indiation of types.

Scalars are handled as such:

```
1  float x,y;
2  comm.bcast( 0,x ); // note: root first
3  comm.allreduce( mpl::plus<float>(), x,y ); // op first
```

where the reduction function needs to be compatible with the type of the buffer.

If your buffer is a `std::vector` you need to take the `.data()` component of it:

```
1  vector<float> xx(2),yy(2);
2  comm.allreduce( mpl::plus<float>(),
3      xx.data(), yy.data(), mpl::contiguous_layout<float>(2) );
```

The `contiguous_layout` is a 'derived type'; this will be discussed in more detail elsewhere (see note 63 and later). For now, interpret it as a way of indicating the count/type part of a buffer specification.

MPL point-to-point routines have a way of specifying the buffer(s) through a *begin* and *end* iterator.

```
// sendrange.cxx
vector<double> v(15);
  comm_world.send(v.begin(), v.end(), 1); // send to rank 1
  comm_world.recv(v.begin(), v.end(), 0); // receive from rank 0
```

Not available for collectives.

The usual reduction operators are given as templated operators:

```
float
  xrank = static_cast<float>( comm_world.rank() ),
  xreduce;
// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
// in place
comm_world.allreduce(mpl::plus<float>(), xrank);
```

Note the parentheses after the operator. Also note that the operator comes first, not last.

Available: `max`, `min`, `plus`, `multiplies`, `logical_and`, `logical_or`, `logical_xor`, `bit_and`, `bit_or`, `bit_xor`.

MPL implementation note: The reduction operator has to be compatible with `T(T,T)>`

TACC

```
template<typename T , typename F >
void mpl::communicator::allreduce
   ( F,const T &, T & ) const;
   ( F,const T *, T *,
     const contiguous_layout< T > & ) const;
   ( F,T & ) const;
   ( F,T *, const contiguous_layout< T > & ) const;
F : reduction function
T : type
```

TACC

```
void mpl::communicator::reduce
    // root, in place
    ( F  f,int  root_rank,T &  sendrecvdata ) const
    ( F  f,int  root_rank,T *  sendrecvdata,const contiguous_layout< T
    // non-root
    ( F  f,int  root_rank,const T &  senddata ) const
    ( F  f,int  root_rank,
            const T * senddata,const contiguous_layout< T > &  l ) cons
    // general
    ( F  f,int  root_rank,const T &  senddata,T &  recvdata ) const
    ( F  f,int  root_rank,
            const T *  senddata,T *  recvdata,const contiguous_layout<
```

TACC

The broadcast call comes in two variants, with scalar argument and general layout:

```
1    template< typename T >
2  void mpl::communicator::bcast
3      ( int root_rank, T &data ) const;
4  void mpl::communicator::bcast
5      ( int root_rank, T *data, const layout< T > &l ) const;
```

Note that the root argument comes first.

```
template<typename T >
void mpl::communicator::bcast
   ( int   root, T &  data ) const
   ( int   root, T *  data, const layout< T > &  l ) const
```

Gathering (by `communicator::gather`) or scattering (by
`communicator::scatter`) a single scalar takes a scalar argument and a raw
array:

```
1  vector<float> v;
2  float x;
3  comm_world.scatter(0, v.data(), x);
```

If more than a single scalar is gathered, or scattered into, it becomes
necessary to specify a layout:

```
1  vector<float> vrecv(2),vsend(2*nprocs);
2  mpl::contiguous_layout<float> twonums(2);
3  comm_world.scatter
4    (0, vsend.data(),twonums, vrecv.data(),twonums );
```

TACC

There is a separate variant for non-root usage of rooted collectives:

```cpp
// scangather.cxx
if (procno==0) {
  comm_world.reduce
    ( mpl::plus<int>(),0,
      my_number_of_elements,total_number_of_elements );
} else {
  comm_world.reduce
    ( mpl::plus<int>(),0,my_number_of_elements );
}
```

TACC

Logically speaking, on every nonroot process, the gather call only has a send buffer. MPL supports this by having two variants that only specify the send data.

```
if (procno==0) {
  vector<int> size_buffer(nprocs);
  comm_world.gather
    (
      0,my_number_of_elements,size_buffer.data()
    );
} else {
  /*
   * If you are not the root, do versions with only send buffers
   */
  comm_world.gather
    ( 0,my_number_of_elements );
```

TACC

```
void mpl::communicator::gather
    ( int  root_rank, const T & senddata ) const
    ( int  root_rank, const T & senddata, T *  recvdata ) const
    ( int  root_rank, const T * senddata, const layout< T > &  sendl ) c
    ( int  root_rank, const T * senddata, const layout< T > &  sendl,
                      T * recvdata, const layout< T > &  recvl ) c
```

Reduce in place

TACC

The in-place variant is activated by specifying only one instead of two buffer arguments.

```
1  float
2    xrank = static_cast<float>( comm_world.rank() ),
3    xreduce;
4  // separate recv buffer
5  comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
6  // in place
7  comm_world.allreduce(mpl::plus<float>(), xrank);
```

Reducing a buffer requires specification of a `contiguous_layout`:

```
1  // collectbuffer.cxx
2  float
3    xrank = static_cast<float>( comm_world.rank() );
4  vector<float> rank2p2p1{ 2*xrank,2*xrank+1 },reduce2p2p1{0,0};
5  mpl::contiguous_layout<float> two_floats(rank2p2p1.size());
6  comm_world.allreduce
7    (mpl::plus<float>(),
       ↪rank2p2p1.data(),reduce2p2p1.data(),two_floats);
8  if ( iprint )
9    cout << "Got: " << reduce2p2p1.at(0) << ","
              << reduce2p2p1.at(1) << endl;
```

reduce2p2p1.at(1)

TACC

The size/displacement arrays for `MPI_Gatherv` / `MPI_Alltoallv` are handled through a `layouts` object, which is basically a vector of *layout* objects.

```cpp
mpl::layouts<int> receive_layout;
for ( int iproc=0,loc=0; iproc<nprocs; iproc++ ) {
  auto siz = size_buffer.at(iproc);
  receive_layout.push_back
      ( mpl::indexed_layout<int>( {{ siz,loc }} ) );
  loc += siz;
}
```

As in the C/F interfaces, MPL interfaces to the scan routines have the same calling sequences as the 'Allreduce' routine.

- Let each process compute a random value n_{local}, and allocate an array of that length. Define

$$N = \sum n_{\mathrm{local}}$$

- Fill the array with consecutive integers, so that all local arrays, laid end-to-end, contain the numbers $0 \cdots N - 1$. (See figure **??**.)

Take the code from exercise 1 and extend it to gather all local buffers onto rank zero. Since the local arrays are of differing lengths, this requires `MPI_Gatherv`.

How do you construct the lengths and displacements arrays?

Arithmetic: `plus`, `multiplies`, `max`, `min`

Logic: `logical_and`, `logical_or`, `logical_xor`.

Bitwise: `bit_and`, `bit_or`, `bit_xor`.

A user-defined operator can be a templated class with an `operator()`.
Example:

```
1  // reduceuser.cxx
2  template<typename T>
3  class lcm {
4  public:
5    T operator()(T a, T b) {
6      T zero=T();
7      T t((a/gcd(a,b))*b);
8      if (t<zero)
9        return -t;
10     return t;
11   }
```

```
1  comm_world.reduce(lcm<int>(), 0, v, result);
```

(The templated class can be a lambda expression)

TACC

You can also do the reduction by lambda:

```
1   comm_world.reduce
2      ( [] (int i,int j) -> int
3          { return i+j; },
4           0,data );
```

Nonblocking collectives have the same argument list as the corresponding blocking variant, except that instead of a *void* result, they return an *irequest*. (See 53)

```
// ireducescalar.cxx
float x{1.},sum;
auto reduce_request =
  comm_world.ireduce(mpl::plus<float>(), 0, x, sum);
reduce_request.wait();
if (comm_world.rank()==0) {
  std::cout << "sum = " << sum << '\n';
}
```

Part III

Point-to-point communication

TACC

- Data type is templated: derived by the compiler.
- Count > 1 is declared in the datatype.

MPL uses a default value for the tag, and it can deduce the type of the buffer. Sending a scalar becomes:

```cxx
// sendscalar.cxx
if (comm_world.rank()==0) {
  double pi=3.14;
  comm_world.send(pi, 1);  // send to rank 1
  cout << "sent: " << pi << '\n';
} else if (comm_world.rank()==1) {
  double pi=0;
  comm_world.recv(pi, 0);  // receive from rank 0
  cout << "got : " << pi << '\n';
}
```

(See also note 19.)

TACC

MPL can send *static array*s without further layout specification:

```
1  // sendarray.cxx
2  double v[2][2][2];
3    comm_world.send(v, 1);  // send to rank 1
4    comm_world.recv(v, 0);  // receive from rank 0
```

Sending vectors uses a general mechanism:

```
1  // sendbuffer.cxx
2  std::vector<double> v(8);
3  mpl::contiguous_layout<double> v_layout(v.size());
4    comm_world.send(v data(), v_layout, 1);  // send to rank 1
5    comm_world.recv(v data(), v_layout, 0);  // receive from rank 0
```

(See also note 20.)

```
template<typename T >
void mpl::communicator::send
   ( const T scalar&,int dest,tag = tag(0) ) const
   ( const T *buffer,const layout< T > &,int dest,tag = tag(0) ) const
   ( iterT begin,iterT end,int dest,tag = tag(0) ) const
T : scalar type
begin : begin iterator
end : end iterator
```

```
template<typename T >
status mpl::communicator::recv
   ( T &,int,tag = tag(0) ) const inline
   ( T *,const layout< T > &,int,tag = tag(0) ) const
   ( iterT  begin,iterT  end,int  source, tag  t = tag(0) ) const
```

Tag types

Tag are `int` or an `enum` typ:

```
1    template<typename T >
2  tag_t (T t);
3  tag_t (int t);
4
```

Example:

```
1  // inttag.cxx
2  enum class tag : int { ping=1, pong=2 };
3  int pinger = 0, ponger = world.size()-1;
4  if (world.rank()==pinger) {
5    world.send(x, 1, tag::ping);
6    world.recv(x, 1, tag::pong);
7  } else if (world.rank()==ponger) {
8    world.recv(x, 0, tag::ping);
9    world.send(x, 0, tag::pong);
10 }
```

MPL differs from other Application Programmer Interfaces (APIs) in its treatment of tags: a tag is not directly an integer, but an object of class tag.

```
1  // sendrecv.cxx
2  mpl::tag t0(0);
3  comm_world.sendrecv
4    ( mydata,sendto,t0,
5      leftdata,recvfrom,t0 );
```

The tag class has a couple of methods such as `mpl::tag::any()` (for the `MPI_ANY_TAG` wildcard in receive calls) and `mpl::tag::up()` (maximal tag, found from the `MPI_TAG_UB` attribute).

The constant `mpl::any_source` equals `MPI_ANY_SOURCE` (by `constexpr`).

The send-recv call in MPL has the same possibilities for specifying the send and receive buffer as the separate send and recv calls: scalar, layout, iterator. However, out of the nine conceivably possible routine signatures, only the versions are available where the send and receive buffer are specified the same way. Also, the send and receive tag need to be specified; they do not have default values.

```
// sendrecv.cxx
mpl::tag t0(0);
comm_world.sendrecv
  ( mydata,sendto,t0,
    leftdata,recvfrom,t0 );
```

The `mpl::status_t` object is created by the receive (or wait) call:

```
mpl::contiguous_layout<double> target_layout(count);
mpl::status_t recv_status =
    comm_world.recv(target.data(),target_layout, the_other);
recv_count = recv_status.get_count<double>();
```

The status object can be queried:

```
1  int source = recv_status.source();
```

The `get_count` function is a method of the status object. The argument type is handled through templating:

```cpp
// recvstatus.cxx
double pi=0;
auto s = comm_world.recv(pi, 0);   // receive from rank 0
int c = s.get_count<double>();
std::cout << "got : " << c << " scalar(s): " << pi << '\n';
```

Nonblocking routines have an `irequest` as function result. Note: not a parameter passed by reference, as in the C interface. The various wait calls are methods of the `irequest` class.

```
1  double recv_data;
2  mpl::irequest recv_request =
3     comm_world.irecv( recv_data sender );
4  recv_request.wait();
```

You can not default-construct the request variable:

```
1  // DOES NOT COMPILE:
2  mpl::irequest recv_request;
3  recv_request = comm.irecv( ... );
```

This means that the normal sequence of first declaring, and then filling in, the request variable is not possible.

> *MPL implementation note:* *The wait call always returns a* `status` *object; not assigning it means that the destructor is called on it.*

Instead of an array of requests, use an `irequest_pool` object, which acts like a vector of requests, meaning that you can *push* onto it.

```
1  // irecvsource.cxx
2  mpl::irequest_pool recv_requests;
3  for (int p=0; p<nprocs-1; p++) {
4    recv_requests.push( comm_world.irecv( recv_buffer[p], p ) );
5  }
```

You can not declare a pool of a fixed size and assign elements.

The `irequest_pool` class has methods `waitany`, `waitall`, `testany`, `testall`, `waitsome`, `testsome`.

The 'any' methods return a $std::pair<bool,size\_t>$, with $false$ meaning $index$==`MPI_UNDEFINED` meaning no more requests to be satisfied.

```
auto [success,index] = recv_requests.waitany();
if (success) {
    auto recv_status = recv_requests.get_status(index);
```

Same for `testany`, then false means no requests test true.

Create two distributed arrays of positive integers. Take the set difference of the two: the first array needs to be transformed to remove from it those numbers that are in the second array.

How could you solve this with an `MPI_Allgather` call? Why is it not a good idea to do so? Solve this exercise instead with a circular bucket brigade algorithm.

Creating and attaching a buffer is done through `bsend_buffer` and a support routine `bsend_size` helps in calculating the buffer size:

```
1  // bufring.cxx
2  vector<float> sbuf(BUFLEN), rbuf(BUFLEN);
3  int size{
       comm_world.bsend_size<float>(mpl::contiguous_layout<float>(BUFLEN))
       };
4  mpl::bsend_buffer buff(size);
5  comm_world.bsend(sbuf.data(),mpl::contiguous_layout<float>(BUFLEN),
       next);
```

Constant: `mpl::bsend_overhead` is *constexpr*'d to the MPI constant `MPI_BSEND_OVERHEAD`.

There is a separate attach routine, but normally this is called by the constructor of the `bsend_buffer`. Likewise, the detach routine is called in the buffer destructor.

```
1   void mpl::environment::buffer_attach (void *buff, int size);
2   std::pair< void *, int > mpl::environment::buffer_detach ();
```

MPL returns a `prequest` from persistent 'init' routines, rather than an `irequest` (MPL note 53):

```
template< typename T >
prequest send_init (const T &data, int dest, tag t=tag(0)) const;
```

Likewise, there is a `prequest_pool` instead of an `irequest_pool` (note 54).

Part IV

Derived Datatypes

MPL mostly handles datatypes through subclasses of the `layout` class. Layouts are MPL routines are templated over the data type.

```cxx
// sendlong.cxx
mpl::contiguous_layout<long long> v_layout(v.size());
  comm.send(v.data(), v_layout, 1); // send to rank 1
```

Also works with complex of float and double.

The data types, where MPL can infer their internal representation, are enumeration types, C arrays of constant size and the template classes `std::array`, `std::pair` and `std::tuple` of the C++ Standard Template Library. The only limitation is, that the C array and the mentioned template classes hold data elements of types that can be sent or received by MPL.

Should you need the `MPI_Datatype` object contained in an MPL layout, there is an access function *native_handle*.

In MPL type creation routines are in the main namespace, templated over the datatypes.

```
// vector.cxx
vector<double>
  source(stride*count);
if (procno==sender) {
  mpl::strided_vector_layout<double>
    newvectortype(count,1,stride);
  comm_world.send
    (source.data(),newvectortype,the_other);
}
```

The commit call is part of the type creation, and freeing is done in the destructor.

The MPL interface makes extensive use of `contiguous_layout`, as it is the main way to declare a nonscalar buffer; see note 20.

TACC

Contiguous layouts can only use predefined types or other contiguous layouts as their 'old' type. To make a contiguous type for other layouts, use `vector_layout`:

```
// contiguous.cxx
mpl::contiguous_layout<int> type1(7);
mpl::vector_layout<int> type2(8,type1);
```

(Contrast this with `strided_vector_layout`; note 66.)

MPL has the `strided_vector_layout` class as equivalent of the vector type:

```
1  // vector.cxx
2  vector<double>
3    source(stride*count);
4  if (procno==sender) {
5    mpl::strided_vector_layout<double>
6      newvectortype(count,1,stride);
7    comm_world.send
8      (source.data(),newvectortype,the_other);
9  }
```

(See note 65 for nonstrided vectors.)

TACC

MPL point-to-point routines have a way of specifying the buffer(s) through a *begin* and *end* iterator.

```
// sendrange.cxx
vector<double> v(15);
  comm_world.send(v.begin(), v.end(), 1);  // send to rank 1
  comm_world.recv(v.begin(), v.end(), 0);  // receive from rank 0
```

Not available for collectives.

Noncontiguous iteratable objects can be send with a `iterator_layout`:

```
1  std::list<int> v(20, 0);
2  mpl::iterator_layout<int> l(v.begin(), v.end());
3  comm_world.recv(&(*v.begin()), l, 0);
```

(See also note 67.)

The templated `subarray_layout` class is constructed from a vector of triplets of global size / subblock size / first coordinate.

```
1  mpl::subarray_layout<int>(
2    { {ny, ny_l, ny_0}, {nx, nx_l, nx_0} }
3  );
```

In MPL, the `indexed_layout` is based on a vector of 2-tuples denoting block length / block location.

```cpp
// indexed.cxx
const int count = 5;
mpl::contiguous_layout<int>
  fiveints(count);
mpl::indexed_layout<int>
  indexed_where{ { {1,2}, {1,3}, {1,5}, {1,7}, {1,11} } };

if (procno==sender) {
  comm_world.send( source_buffer.data(),indexed_where, receiver );
} else if (procno==receiver) {
  auto recv_status =
    comm_world.recv( target_buffer.data(),fiveints, sender );
  int recv_count = recv_status.get_count<int>();
  assert(recv_count==count);
}
```

TACC

For the case where all block lengths are the same, use
`indexed_block_layout`:

```
// indexedblock.cxx
mpl::indexed_block_layout<int>
  indexed_where(   1, {2,3,5,7,11} );
comm_world.send( source_buffer.data(),indexed_where, receiver );
```

One could describe the MPI struct type as a collection of displacements, to be applied to any set of items that conforms to the specifications. An MPL `heterogeneous_layout` on the other hand, incorporates the actual data. Thus you could write

```
// structscalar.cxx
char c; double x; int i;
if (procno==sender) {
  c = 'x'; x = 2.4; i = 37; }
mpl::heterogeneous_layout object( c,x,i );
if (procno==sender)
  comm_world.send( mpl::absolute,object,receiver );
else if (procno==receiver)
  comm_world.recv( mpl::absolute,object,sender );
```

Here, the `absolute` indicates the lack of an implicit buffer: the layout is absolute rather than a relative description.

TACC

More complicated data than scalars takes more work:

```cpp
// struct.cxx
char c; vector<double> x(2); int i;
if (procno==sender) {
  c = 'x'; x[0] = 2.7; x[1] = 1.5; i = 37; }
mpl::heterogeneous_layout object
  ( c,
    mpl::make_absolute(x.data(),mpl::vector_layout<double>(2)),
    i );
if (procno==sender) {
  comm_world.send( mpl::absolute,object,receiver );
} else if (procno==receiver) {
  comm_world.recv( mpl::absolute,object,sender );
}
```

Note the `make_absolute` in addition to `absolute` mentioned above.

TACC

Resizing a datatype does not give a new type, but does the resize 'in place':

```
1   void layout::resize(ssize_t lb, ssize_t extent);
```

Part V

Communicator manipulations

Code:

```cpp
const mpl::communicator &comm =
  mpl::environment::comm_world();
MPI_Comm
  world_extract = comm.native_handle(),
  world_given = MPI_COMM_WORLD;
int result;
MPI_Comm_compare(world_extract,world_given,&result);
cout << "Compare raw comms: " << "\n"
         << "identical: " <<
     (result==MPI_IDENT)
         << "\n"
         << "congruent: " <<
     (result==MPI_CONGRUENT)
         << "\n"
         << "unequal  : " <<
     (result==MPI_UNEQUAL)
         << "\n";
```

Output:

```
Compare raw comms:
identical: true
congruent: false
unequal  : false
```

MPL does not allow for access to the wrapped communicators. However, for `MPI_COMM_WORLD`, the routine `MPI_Comm_set_errhandler` can be called directly.

In MPL, splitting a communicator is done as one of the overloads of the communicator constructor;

```
1  // commsplit.cxx
2  // create sub communicator modulo 2
3  int color2 = procno % 2;
4  mpl::communicator comm2( mpl::communicator::split, comm_world, color2
       ↪ );
5  auto procno2 = comm2.rank();
6
7  // create sub communicator modulo 4 recursively
8  int color4 = procno2 % 2;
9  mpl::communicator comm4( mpl::communicator::split, comm2, color4 );
10 auto procno4 = comm4.rank();
```

> MPL implementation note: The `communicator::split` identifier is an object of class `communicator::split_tag`, itself is an otherwise empty subclass of `communicator`:
>
> ```
> 1  class split_tag {};
> 2  static constexpr split_tag split{};
> ```

TACC

Similar to ordinary communicator splitting 78:
`communicator::split_shared`.

Part VI

Process topologies

The constructor `dist_graph_communicator`

```
1   dist_graph_communicator
2       (const communicator &old_comm, const source_set &ss,
3        const dest_set &ds, bool reorder = true);
```

is a wrapper around `MPI_Dist_graph_create_adjacent`.

Methods `indegree`, `outdegree` are wrappers around
`MPI_Dist_graph_neighbors_count`. Sources and targets can be queried with
`inneighbors` and `outneighbors`, which are wrappers around
`MPI_Dist_graph_neighbors`.

Part VII

Other

The timing routines `wtime` and `wtick` and `wtime_is_global` are environment methods:

```
double  mpl::environment::wtime ();
double  mpl::environment::wtick ();
bool mpl::environment::wtime_is_global ();
```

MPL always calls `MPI_Init_thread` requesting the highest level
`MPI_THREAD_MULTIPLE`.

```
1  enum mpl::threading_modes {
2    mpl::threading_modes::single = MPI_THREAD_SINGLE,
3    mpl::threading_modes::funneled = MPI_THREAD_FUNNELED,
4    mpl::threading_modes::serialized = MPI_THREAD_SERIALIZED,
5    mpl::threading_modes::multiple = MPI_THREAD_MULTIPLE
6  };
7  threading_modes mpl::environment::threading_mode ();
8  bool mpl::environment::is_thread_main ();
```

TACC

MPL is not a full MPI implementation

- File I/O
- One-sided communication
- Shared memory
- Process management