UNIVERSITÀ DI PISA

The MPI Message-passing Standard
Practical use and implementation (V-VI)

SPD Course

6/03/2026

Massimo Coppola

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

HPC lab

Topics covered

- (Blocking) Collective Communications in Intracommunicators
  - brief mention to non-blocking collectives
- Refining Derived Datatypes Layout for composition
- Collective Primitives involving Communication and Computation

Intracommunicators

BLOCKING COLLECTIVE COMMUNICATIONS

Characteristics of Collectives

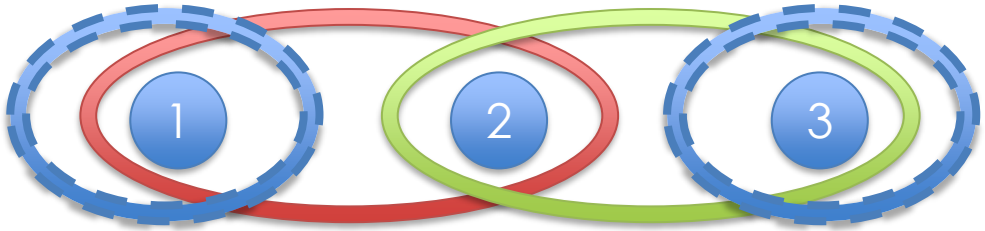
- Collective operations are called by ALL processes of a communicator
  - Still happen within a communicator like point-to-point, but isolated from them
  - Use Datatypes to define message structure
  - Implement complex communication patterns
- Distinct semantics from point-to-point
  - No modes
  - *Always blocking* (* MPI 3 changed this *)
  - No unmatched variable-size data
  - No status parameters (would require many…)
  - Limited concurrency
- Still a lot of freedom left to implementers
  - E.g. actual pattern choice, low-level operations
  - Semantics carefully defined for this aim

Collective & Communicators

- Independence among separate communicators
- Independence with any point-to-point in the same communicator
  - Although collectives may be implemented on top of p-to-point, e.g. by using a separate set of tags
- Collectives are serialized over a communicator
  - Obvious consequence of the semantics
  - Collectives must share the same actual call order from every process in the communicator
- Serialization **is not** synchronization
  - Blocking behaviour = after the call, local completion is granted and buffer / parameters are free to be reused
  - Globally, the collective may still be ongoing (and vice versa)
  - Example: broadcast on a binary support tree may complete on root process long before it is done
  - p-to-point primitives are concurrent with collective operations
  - **Only** MPI_Barrier is granted to synchronize
- Serialization **is** a source of deadlocks

Example of deadlocks and errors

- Serialization **is** a source of deadlocks
  - 3 overlapping comm.s with blocking collectives in conflicting order



BAR BRD BAR

BRD BAR BAR

OK

BAR BAR BAR

BAR BAR BAR

Deadlock!

Collective Primitives – High-level view

- Many of the primitives you already know
  - Synchronization: Barrier (*also an all-to-all*)
  - One-to-all: Bcast (*broadcast*), Scatter *
  - All-to-one: Gather *, Reduce
  - All-to-all: AllGather *, AllToAll *, AllReduce, ReduceScatter
  - Other (*computational-communication patterns* and *management primitives*): Scan (*parallel prefix*), Exscan *Communicator-building* operations

Collectives: Semantics

- All processes send and/or receive data
  - If a structure is distributed, one piece is possibly sent/received by the same process
  - This in general includes the root process, if one is present
  - Semantics are symmetric to simplify the case where the root process dynamically changes at runtime
- Agreement on parameters among all processes
  - Which process is the root, if a root role is needed
  - Specific roles in communicator building, operators in computational collective
- Agreement on data to be transferred
  - Buffers defined at each process must match in size and type signature with what is required by the partner sending/receiving that data
    - **Even if the actual communication may happen differently!**
  - In some cases the same buffer is used for reading AND writing

Collectives: Semantics

- User-defined datatypes and type signatures are allowed
  - However, more constraints than in the p-to-p case
  - Type signatures should be compatible as always
  - *Writing* typemaps shall never be redundant
    - No ambiguity shall ever arise from typemap access order, which is free choice of the MPI library
  - Generally speaking, collective primitives should not read or write twice the same location
    - no location written twice by either the same or different processes inside a collective
    - can imply that no location is even **read** twice
    - **Not** discussing all cases, refer to the standard

Barrier & Broadcast

- `int MPI_Barrier(MPI_Comm comm)`
  - can be applied to intercommunicators
  - the only collective whose synchronization effects are guaranteed by the MPI standard

- `int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )`
  - semantics: the specified communication is sent to all processes
    - equivalent descriptions always given in the standard
  - can use any underlying scheme (trivial, n-ary tree, spanning tree...)

Classifications of collectives

- MPI has plenty of distinct collective comm. calls
  - Distinct == a different API function name and signature
  - 17 blocking and 17 non-blocking, + some more for communicator management

1. Classification by asymmetry
   - All to 1 many processes send to one
   - 1 to All one process sends to many
   - All to All all processes send and receive
2. by homogeneity of data exchange
   - "*normal*" = homogeneous communications
   - V "*variable*" = a count/size for each communication is specified by the process
3. By kind of pattern
   - Communication only
   - Communication **and** Computation (A-to-1, A-to-A)

Collectives and Inter-communicators

- All processes in the intercom. (both groups) must call each collective op.
- Collective semantics is related to the collective class
  - One-to-all, All-to-one : unidirectional data transfer, actual sender specified by the argument *root* (MPI_ROOT vs MPI_PROC_NULL)
  - All-to-all: bidirectional data transfer among the two groups
  - Semantics is specified case by case
  - in-place communication disallowed
  - see MPI 5.0 sec. 6.2.3, and special cases of collective definitions in chapter 6

Gather

```
int MPI_Gather(
        const void* sbuf, int scount, MPI_Datatype sendtype,
        void* recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```

- All to 1
  - gather a distributed data structure at the root process
- the send and recv type signatures must match
  - like a couple of point-to-point communication
  - all send specs must match the recv at the root
- the actual recv buffer and data structure is N times bigger than the recv specification
  - where N is the number of processes in comm
- process rank *i* will write at position *i* of this buffer
  - exact address is recvbuf+i*count*mpi_size(recvtype)
- the receive buffer count and type is significant only at the root
  - it is ignored on other processes
  - the root can use MPI_IN_PLACE for the send buffer

in-place Communication

- In collectives, all processes send or receive data, **including** the designed root
  - much like a send or receive to MPI_PROC_SELF
  - this means extra work and extra buffers
- MPI_IN_PLACE constant
  - to be specified as a buffer address
  - specifies that the input and output buffers at this process for this collective are the same
  - to be used as the send or receive buffer, depending on the collective
  - the associated count, datatype parameters are ignored
- why?
  - explicitly avoid useless data movement
  - simplify usage of collectives in many common cases (less parameters needed and less error prone)
  - avoid the limitation of languages that forbid aliasing of parameters (e.g. Fortran)
- not allowed within intercommunicators

Scatter

```
int MPI_Scatter(
        const void* sendbuf, int sendcount, MPI_Datatype sendtype,
        void* recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```

- 1 to All
  - scatter a data structure from the root process onto the whole comm
- the send and recv type signatures must match
  - like a couple of point-to-point communication
  - all send specs must match the recv at the root
- the actual send buffer and data structure is N times bigger than the send specification
  - where N is the number of processes in comm
- process rank *i* will read from at position *i* of this buffer
  - exact address is sendbuf+i*count*mpi_size(sendtype)
- the send buffer count and type are significant only at the root, and ignored on other processes
  - the root can use MPI_IN_PLACE for the recv buffer

Gatherv = Gather Variable-length

- ```
  int MPI_GatherV(
      const void* sbuf, int scount, MPI_Datatype sendtype,
      void* recvbuf, const int recvcounts[],
      const int displs[], MPI_Datatype recvtype,
      int root, MPI_Comm comm)
  ```

- All-to-one
- like Gather, but the parts of the gathered structure can be a different size each one
  - the receive count is now an array of integers
  - the send counts can vary, communications sizes are no longer bound to be the same on all processes
  - some counts can be zero
- also: place in memory for received parts is given
  - process of rank *i* will write at position displs[*i*]*mpi_extent (recvtype) of recvbuf
  - the order of the received parts can be arbitrarily changed
- the send and recv type signatures must **still** match on each couple of processes
  - more complex to check, but no real change
  ```

Variable-length : Scatterv

```
int MPI_Scatterv(
        const void* sendbuf, const int sendcounts[], const int displs[],
        MPI_Datatype sendtype,
        void* recvbuf, int recvcount, MPI_Datatype recvtype,
        int root, MPI_Comm comm)
```

- Analogous to the variable-length gather, but performing a scatter

Allgather

```
int MPI_Allgather(
      const void* sendbuf, int sendcount, MPI_Datatype sendtype,
      void* recvbuf, int recvcount, MPI_Datatype recvtype,
      MPI_Comm comm)
```

- Same semantics of gather, but all processes actually perform the gather operation and get the result (no root process specification)
- Semantics is the same as gather + broadcast, but the communication pattern may be optimized by MPI
- Also has a V form, **MPI_Allgatherv**

MPI_ALLTOALL

```
int MPI_Alltoall(
        const void* sendbuf, int sendcount, MPI_Datatype sendtype,
        void* recvbuf, int recvcount, MPI_Datatype recvtype,
        MPI_Comm comm)
```

- Further generalized communication, each process sends distinct data to all other processes
- All blocks of data have the same definition

MPI_ALLTOALLV

```
int MPI_Alltoallv(
      const void* sendbuf, const int sendcounts[],
      const int sdispls[], MPI_Datatype sendtype,
      void* recvbuf, const int recvcounts[],
      const int rdispls[], MPI_Datatype recvtype,
      MPI_Comm comm)
```

- Further generalized communication, each process sends distinct data in different amount to all other processes
- **MPI_Alltoallw** further generalizes the pattern, also allowing distinct receive and send datatypes for each distinct communication portion among a couple of processes

Changes since MPI 3.0

- Collective Communications **can** also be non-blocking
  - In this course we will mainly focus on the blocking version
- After studying the blocking version, non-blocking collectives are summarized as
  - names gain an "I" e.g. MPI_BCAST → MPI_IBCAST
  - blocking and non-blocking collectives do not match with each other
  - completion checked via all {WAIT * , TEST *} calls on all participating processes
  - multiple outstanding collectives are allowed in same communicator
  - non-blocking behavior can avoid collective-related deadlock across communicators
    - interaction with collective serialization **is** significant
  - it is not allowed to cancel a non-blocking collective

Datatypes

REFINING DERIVED DATATYPES LAYOUT FOR COMPOSITION

Derived D.type Extent and composition

- For complex derived datatypes, extent plays an important role
  - Plain definition : distance between the first (smallest address) byte and last (largest address) byte used in memory
  - Actual use : the offset between two items of the given datatype when they are stored consecutively in memory
    - E.g. whenever a contiguous datatype is created or a communication buffer with more instances is used
- Setting extent manually (MPI1, MPI2>)
- Querying extent
- Examples with derived datatypes

Get extent

```
Int MPI_Type_get_extent_x (MPI_Datatype datatype,
      MPI_Count *lb, MPI_Count *extent)
```

- Get the lower bound and extent of a datatype
  - By default, lower bound = lowest-address location of a datatype
  - Extent = distance from lower bound to highest address location used by the datatype

Modify extent

```
int MPI_Type_create_resized(
      MPI_Datatype oldtype,
      MPI_Aint lb, MPI_Aint extent,
      MPI_Datatype *newtype)
```

- Modify the lower bound and extent of a datatype
- Reset lower bound and extent of the datatype to new arbitrary values, for the sake of data structures composition

Retrieve original extent

```
int MPI_Type_get_true_extent(MPI_Datatype datatype,
      MPI_Aint *true_lb, MPI_Aint *true_extent)
```

- Retrieve the true lower bound and extent values from a datatype
- MPI always keeps the information as it is needed for the actual packing and unpacking operations

Intracommunicators

COLLECTIVE PRIMITIVES WITH COMMUNICATION AND COMPUTATION

Reduce

- ```
  int MPI_Reduce(
      const void* sendbuf, void* recvbuf, int count,
      MPI_Datatype datatype,
      MPI_Op op, int root, MPI_Comm comm)
  ```

- reduce operation across all processes of a Communicator
  - Reduces the elements in the same position of each process' buffer, leaving results in root's buffer
- count, datatype, op, root, comm arguments must match
  - If count == 1 we have a classical element-wise reduction
  - If count>1 we have several reductions at the same time
- As with any collective, the communication pattern is implementation dependent (but is op commutative ?)
- MPI provides most basic operators
  - Operators are associative
  - Operators may be commutative → potential optimizations
  - Note that: floating point op.s may be seen as non-commutative
  - *Datatype* must be compatible with *op*
  ```

MPI Scan/Reduce operators

- Arithmetic operations
  - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD
    - Generally allowed on MPI integral and floating point types (including complex)
- Logic (L) and bit wise (B) operations
  - MPI_LAND, MPI_LOR, MPI_LXOR
    - Generally allowed on C integers and on logical types
  - MPI_BAND, MPI_BOR, MPI_BXOR
    - Generally allowed on C/Fortran integers

MPI_MINLOC and MAXLOC

- Operators defined on couples (value, index)
  - MPI_MAXLOC, MPI_MINLOC
  - Value is any integral or floating point type
  - Index is an integral type
  - chars used as integers require special attention
    - e.g. explicitly using MPI_SIGNED _CHAR / UNSIGNED_CHAR
- MINLOC : compute the global minimum of v and the index attached to it
- MAXLOC : compute the global maximum of v and the index attached to it
- Lexicographic order
  - when more values hit the minimum (maximum) the lower one is chosen
- Example application
  - pass (value, rank) to detect the rank of the process with the minimum/maximum value

MPI couple types

- These couple types are supported by the MPI_MINLOC and MPI_MAXLOC operators

- MPI_FLOAT_INT - struct { float, int }
- MPI_LONG_INT - struct { long, int }
- MPI_DOUBLE_INT - struct { double, int }
- MPI_SHORT_INT - struct { short, int }
- MPI_2INT - struct { int, int }
- MPI_LONG_DOUBLE_INT
  - struct { long double, int }
  - this is an OPTIONAL type

Operator semantics

- Operators are called within the reduction collective by the instances of the MPI library of the processes of the program
- Each operators receives two local buffers and performs a reduction step on their contents
  - The buffers are possibly allocated by the library implementation as temporaries
  - Many operators are polymorphic, so they have to detect the type of data in the buffer
  - Datatype is a parameter passed from the collective down to the operator, but remember it is a handle
    - Easy case: MPI basic datatypes are globally known to MPI runtime and to the program
    - Besides, MPI standard operators are easy

MPI operators and computing-collective primitives

- MPI operators (including user-defined ones) are used by all MPI collectives performing distributed computation
  - MPI_REDUCE, MPI_ALLREDUCE, MPI_REDUCE_SCATTER_BLOCK, MPI_REDUCE_SCATTER, MPI_SCAN, MPI_EXSCAN
  - All the non blocking version of those collectives (since MPI-3)
  - MPI_REDUCE_LOCAL (special case actually designed for MPI implementers)

User-defined operators

- MPI allows you to define your own operators
  - They can apply to basic and user-defined datatypes
- What do you need to do
  - (Possibly) provide relevant datatype definitions
  - Provide MPI with a definition of the operator
    - a compiled function with a specific signature
    - the operator definition this is local to each process
  - Detect and recognize the MPI_Datatype within the operator code
    - To detect errors
    - If the operator needs to be polymorphic
  - Combine each couple of elements in the same position of the two input buffers
- Operator code can call **no** MPI communication primitives; only MPI_ABORT() in case of error

MPI_Op_create

- ```
  int MPI_Op_create(MPI_User_function* user_fn,
          int commute, MPI_Op* op)
  ```

- MPI primitive for defining operators
- Takes a user function pointer as first argument
- Can specify non-commuting operators
- Returns the operator handle

- MPI_Op_free allows to free operators
  ```

Operator signature

- ```
  typedef void MPI_User_function(
          void* invec,
          void* inoutvec,
          int *len,
          MPI_Datatype *datatype);
  ```

- row-wise combines data from two buffers
  - results are placed in the second buffer
- The datatype handle comes from the collective call (e.g. reduction) and may be unknown at compile time
  - For user-defined datatypes, polymorphic operators need to access a table of datatypes handles that are defined by the program
  ```

Example : rewriting MPI_SUM

```
/* this follows the MPI_User_function typedef */
void my_sum_op(void * b_in, void * b_inout,
               int * count, MPI_Datatype * d) {
    if (d == MPI_INT) {
        for (i=0; i<count; i++) {
            ((int*)b_inout)[i]+=((int *)b_in)[i];  }
    } else if (d == MPI_FLOAT) {
        for (i=0; i<count; i++) {
            ((float*)b_inout)[i]+=((float *)b_in)[i];}
    } else MPI_Abort (MPI_COMM_WORLD, -12345);
}

... ...

MPI_Op * op_sum;
MPI_Create_op (* my_sum_op, MPI_FALSE, op_sum)
```

- Very limited example: it only accepts INT and FLOAT types
- Can call specialized functions in each case (code reuse, hardware acceleration)

Datatypes and polymorphic operators

- Check the datatype
  - Compare the received datatype handle to a list of allowed handles, execute proper code
  - Simple if/else error if only one type is allowed
- Check and switch for polymorphic op.s
  - Operators that can handle several datatypes should employ data structures that avoid any excessive comparison overhead
    - E.g. a hash-map (perfect hash?) associating handles with code (function pointers) implementing each case of use of the operator
  - The overhead is usually negligible with respect to the communication overhead of a reduction or scan

Example : rewriting MPI_SUM (II)

```
/* this follows the MPI_User_function typedef */
void my_sum_op(void * b_in, void * b_inout,
               int * count, MPI_Datatype * d) {
    int my_type = my_hashtable_get(d);
    case (SUMOP_INT_T) : // MPI_INT
        sumintarrays((int *)b_in, (int*) b_inout, count);
        break;
    case (SUMOP_FLOAT_T) : // MPI_FLOAT
        sumfloatarrays((float *) b_in, (float *) b_inout, count);
            break;
    case (SUMOP_4BY4_FLOAT_T) : // user type example, 4*4 matrix
        sumfloatarrays((float *)b_in, (float*)b_inout, count*16);
        break;
    default : MPI_Abort (MPI_COMM_WORLD, -12345);
}
```

- In the example we assume that
  - a hashtable is filled with custom values for each recognized datatype
  - if the type is not in the hashtable, the default value returned (*unknown datatype*) triggers the default case

Scan

- `int MPI_Scan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

- Applies a scan (parallel prefix) to the elements in corresponding position of the send buffers of the processes
- The scan works according to process rank
  - Process i computes the result of the combination of data from processes { 0, .. i }
  - Scan is the identity for process with rank 0
- If count>1 we have multiple scans within the same communication pattern
- With MPI_INPLACE in sendbuf, only the receive buffer is used

ExScan

- `int MPI_Exscan(const void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

- Same as MPI_Scan, but results are accumulated on the *following* process
  - Process with rank i gets the parallel prefix result of data contributed from processes 0.. i-1
  - Mnemonics: think of it as a “scan and shift”
  - Process 0 receives no data, and its receive buffer is not used by MPI_Exscan
  - MPI_IN_PLACE can be used in sendbuf

Other reduce collective operations

- MPI_Allreduce
  - Semantically equivalent to a reduce followed by a broadcast
  - May be implemented more efficiently, of course
- MPI_Reduce_scatter_block
  - Performs a reduction, then scatters the result buffer across the processes
  - Requires n*recvcount elements by each process, scatters the n blocks of recvcount elements of the result
  - Parameter recvcount is the number of elements received per process after the scatter
    - **the overall reduction is computed on recvcount*N elements, where N is the communicator size.**
- MPI_Reduce_scatter
  - Generalizes the scatter_block to a variable scatter (each process can receive a block of different size)
  - The recvcount is now an array of block sizes (the array is the same size as the communicator, see MPI_Scatterv)

References

- **With respect to MPI-5 standard**
  - Chapter 6 (Global reduction operators)
    - sec. 6 – 6.8 for communication collectives
  - Skip intercommunicator collectives, at least on first reading (sec. 6.2.2, 6.2.3 and related parts of each collective specification)
  - You can skip reduce_local
  - sec. 6.9 – 6.11 for collective reduction and computing collectives
    - sec. 6.9.5 addresses custom reduction operators
  - we did not address nonblocking collectives (sec. 6.12)
  - we did not address persistent collectives (sec. 6.13)