

The MPI Message-passing Standard

Practical use and implementation (V)

SPD Course

16/03/2018

Massimo Coppola

Intracommunicators

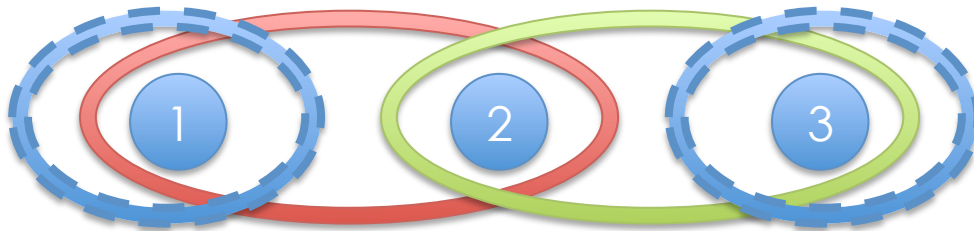
COLLECTIVE COMMUNICATIONS

- Collective operations are called by ALL processes of a communicator
 - Still happen within a communicator like p-to-p
 - Use Datatypes to define message structure
 - Implement complex communication patterns
- Distinct semantics from point-to-point
 - No modes
 - Always blocking (* MPI 3 changes 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

- Independence among separate communicators
- Independence with any p-to-point in same comm.
 - 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 op.s
 - **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 collectives in conflicting order



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
- More on this later on

- 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

- 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 either **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-3 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)

- **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, an 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)

- **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)`
 - like Gather, but the parts of the gathered structure are allowed to 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 $\text{displs}[i] * \text{mpi_extent}(\text{recvtype})$ 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`**



- `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

- **MPI standard 3.0 released in September 2012**
  - Collective Communications **can** be non-blocking
  - In this course we will stick to the MPI 2.2 definition
- **After** studying the blocking version, it might worth to know about non-blocking collectives
  - 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
  - multiple outstanding collectives 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-bl. collective

- MPI standard Relevant Material for 3<sup>rd</sup> lesson
  - Chapter 2:  
sec.
  - Chapter 3:  
sec. 3.2.5, 3.2.6, 3.6, 3.7, 3.11
  - Chapter 4:  
sec. 4.1.2, (skip 4.1.3, 4.1.4) , 4.1.5 – 4.1.7, 4.1.11
  - Chapter 5:  
sec.