Lecture 3: MPI Basics

Communicators & point-to-point communications

MPI 4 standard: https://www.mpi-forum.org/docs/mpi-4.0/mpi40- report.pdf
MPI 3 (version 3.1) standard: https://www.mpi-forum.org/docs/mpi3.1/mpi31-report.pdf
https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf
OpenMPI documentation: https://www.open-mpi.org

- What would be a “minimal” MPI-based program (which does some communication)? (3 min)

```
MPI_Status status;
MPI_Init();
MPI_Sendrecv(&a, 1, MPI_INT, 0, 0, &b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
MPI_Finalize();
```

- What is a group, what is a communicator, what is a context?

  You already know MPI_Send & MPI_Recv.
  Is their "tag" argument a part of the context? (4 min)

- Group: set of numbered processes
- Communicator: group + context + virtual topology [+ attributes]
  = encapsulator (think of libraries)
- Context: communicator-specific labels or tags of messages
  partition the communication space:
  "A message sent in one context cannot be received in another context."
  "Collective operations are independent of point-to-point operations."
- Message tag: message-specific, not part of communicator context

- About send and receive functions (MPI_Send, MPI_Recv & other):
  Which matching rules do exist for their data type parameters?
  Exceptions?
  Which parameters can be wildcarded? (5 min)

- Data types must match
  - between send and receive calls (exception: MPI_PACKED)
  - between caller declarations and types in MPI calls (exceptions: MPI_BYTE , MPI_PACKED)

  data counts need not to match

  (note: receive count="length of receive buffer", not "length of message")

- Wildcards: MPI_ANY_SOURCE, MPI_ANY_TAG

- Receive functions:
  Which information can be obtained from the status parameter? (2 min)

- MPI_SOURCE, MPI_TAG and MPI_ERROR
  indirectly: data count by MPI_GET_COUNT
  often not relevant → use MPI_STATUS_IGNORE

- Point-to-point communication
  Explain the pair of opposites: (10 min)

  blocking – nonblocking
  synchronous – asynchronous
  buffered – unbuffered
  local – nonlocal
  one-sided – two-sided

  When are separate completion calls needed?
  Give examples for such functions!

- Blocking vs. non-blocking: different in function returning
  blocking: returns only when success criterion fulfilled
  non-blocking: returns immediately
- Asynchronous vs. synchronous: different in success criterion
  asynchronous: ``buffer is copied"
  synchronous: ``data are (beginning to be) received" (= handshake w. receiver)
- Buffered vs. unbuffered: different in buffer use (user-provided or system)
  buffered: copies data immediately to buffer for later transmission
  implies asynchronous
- Local vs. non-local: different in dependence on other process
  local: returns irrespective of execution within another process
- One-sided vs. two-sided: different in number of parties involved
  one-sided: only one party (getter or putter) = remote memory access(RMA)
  (but 2nd party needs to enable access)
  two-sided: two parties - corresponding sender and receiver

- Completion routines: needed for non-blocking calls, employ request parameter
- Examples: MPI_WAIT, MPI_TEST

- Fill the table!
  Possibilities: yes/no/depends/not applicable (8 min)

type	blocking	synchronous	buffered	local	remarks
MPI_Bsend	yes	no	yes	yes	
MPI_Ibsend	no	no	yes	yes	
MPI_Ssend	yes	yes	no	no	
MPI_Issend	no	yes	no	yes	
MPI_Send	yes	depends	depends	depends	“standard”
MPI_Isend	no	depends	depends	yes	— ”—
MPI_Rsend	yes	depends	depends	depends	“ready”
MPI_Irsend	no	depends	depends	yes	= receive must be posted
MPI_Sendrecv	yes	depends	depends	no	no deadlocks
MPI_Recv	yes	not applicable	not applicable	no	
MPI_Irecv	no	not applicable	not applicable	yes	

- What means “non-overtaking”?
  Under which conditions does MPI guarantee determinism? (5 min)

  Discuss:

```
if (rank==0) {
  MPI_Bsend(buf1, count, MPI_FLOAT, 1, tag, comm);
  MPI_Bsend(buf2, count, MPI_FLOAT, 1, tag, comm);
} else {
  if (rank==1) {
   MPI_Recv(buf1, count, MPI_FLOAT, 0, MPI_ANY_TAG, comm, status);
   MPI_Recv(buf2, count, MPI_FLOAT, 0, tag, comm, status);
  }
}
```

- Non-overtaking: temporal order of sends is preserved in receiving

- guarantees determinism, conditions:
  - no use of MPI_ANY_SOURCE
  - single-threaded program

- Example code: messages are received in intended order, although both sends match both receives, but receive calls consume message

- Discuss sample code MPI/MPI_SR_1.c!
  Is it safe? How can it be made safe?
- Compare with MPI/MPI_SR_2.c!
  Would using MPI_Rsend here improve things? (4 min)

- MPI/MPI_SR_1.c not safe, success depends on message size
  Safe: by
  - explicit buffering
  - non-blocking sends
  - proper order

- MPI/MPI_SR_2.c always fails (hangs) as blocking receives never return
  MPI_Rsend can't help, non-blocking receives would

- Discuss sample code MPI/MPI_SR_3.c!
  Which version is safe? (3 min)

- not safe, success depends on message size
- different send-receive order for the two processes is safe

- Discuss sample code MPI/MPI_SR_4.c!
  Would it deadlock?
  If not, describe the course of events! (4 min)

- Code doesn’t deadlock
- Execution strictly serial, starting of send by penultimate process

- Discuss sample code MPI/MPI_SR_5.c!
  Is it safe?
  Is it more efficient than the preceding (linear chain)?
  Can it be simplified? (4 min)

- Is safe
- More efficient as comm_size-2 communications executed in parallel
- Calls for first and last process need not to be serial

- What happens in general when replacing standard send by synchronous send, i.e. MPI_Send(<pars>); –> MPI_Ssend(<pars>); ? (4 min)

- If code was safe: no change
- If code was unsafe (relying in implicit buffering): fails