The “Logically Concurrent” Issue

MPI Forum Dec 2021 – Dan Holmes/HACC WG
(now with comments from & after the meeting)

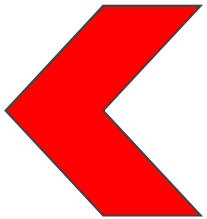
Where is the problem?

3.5 Semantics of Point-to-Point Communication

A valid MPI implementation guarantees certain general properties of point-to-point communication, which are described in this section.

Order Messages are **non-overtaking**: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives. It guarantees that message-passing code is deterministic, if processes are single-threaded and the wildcard MPI_ANY_SOURCE is not used in receives. (Some of the calls described later, such as MPI_CANCEL or MPI_WAITANY, are additional sources of nondeterminism.)

If a process has a single thread of execution, then any two communications executed by this process are **ordered**. On the other hand, if the process is multithreaded, then the semantics of thread execution may not define a relative order between two send operations executed by two distinct threads. The operations are **logically concurrent**, even if one physically precedes the other. In such a case, the two messages sent can be received in any order. Similarly, if two receive operations that are **logically concurrent** receive two successively sent messages, then the two messages can match the two receives in either order.



Single-threaded example

Example 3.6 An example of non-overtaking messages.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
   CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
   CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank .EQ. 1) THEN
   CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
   CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

The message sent by the first send must be received by the first receive, and the message sent by the second send must be received by the second receive.

What about matching rules?

- There is no debate about the meaning of matching rules
  - If using different communicators, then there is no ambiguity
  - If using different explicit tag values at the receiver, then there is no ambiguity
  - If using different source or dest, then there is no ambiguity

- There is no debate about the (single-threaded) example code
  - If both MPI calls a made from the same thread, then there is no ambiguity
  - If the thread support level is MPI_THREAD_SINGLE or MPI_THREAD_FUNNELED, then there is no ambiguity

Multi-threaded example(s)

Missing

Let’s fix that oversight

Multi-threaded example (1st try)

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
   !$OMP PARALLEL DEFAULT(SHARED) THREADPRIVATE(tid, ierr) NUM_THREADS(2)
   !$ tid = OMP_get_thread_num()
   !$ IF (tid .EQ. 0) THEN
      CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
   !$ ELSEIF (tid .EQ. 1) THEN
      CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
   !$ ENDIF
   !$OMP END PARALLEL
ELSEIF (rank .EQ. 1) THEN
   CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
   CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

First try — notes

- This code requires MPI_THREAD_MULTIPLE
  - No synchronisation between threads, concurrent execution
- “Semantics of thread execution [do] not define a relative order”
  - Which send happens first? Which send happens second?
- This code does not match the single-threaded example
  - From a process-centric point-of-view
- This formulation does not cover all multi-threaded cases/situations
  - What about multi-threaded but not MPI_THREAD_MULTIPLE?

Multi-threaded example (2nd try)

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
   !$OMP PARALLEL DEFAULT(SHARED) THREADPRIVATE(tid, ierr) NUM_THREADS(2)
   !$ tid = OMP_get_thread_num()
   !$OMP CRITICAL
   !$ IF (tid .EQ. 0) THEN
      CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
   !$ ELSEIF (tid .EQ. 1) THEN
      CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
   !$ ENDIF
   !$OMP END CRITICAL
   !$OMP END PARALLEL
ELSEIF (rank .EQ. 1) THEN
   CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
   CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

Second try — notes

- This code requires at least MPI_THREAD_SERIALISED
  - Synchronisation between threads prevents concurrent execution
- “Semantics of thread execution [do] not define a relative order”
  - Which send happens first? Which send happens second?
- This code does not match the single-threaded example
  - From a process-centric point-of-view

Multi-threaded example (3rd try)

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .EQ. 0) THEN
   !$OMP PARALLEL DEFAULT(SHARED) THREADPRIVATE(tid, ierr) NUM_THREADS(2)
   !$ tid = OMP_get_thread_num()
   !$ IF (tid .EQ. 0) THEN
      CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
   !$ ENDIF
   !$OMP BARRIER
   !$ IF (tid .EQ. 1) THEN
      CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
   !$ ENDIF
   !$OMP END PARALLEL
ELSEIF (rank .EQ. 1) THEN
   CALL MPI_RECV(buf1, count, MPI_REAL, 0, MPI_ANY_TAG, comm, status, ierr)
   CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

Third try — notes

- This code requires at least MPI_THREAD_SERIALISED
  - Synchronisation between threads prevents concurrent execution
- Semantics of thread execution DO define a relative order
  - We know for certain which send happens first/second
- This code DOES match the single-threaded example
  - From a process-centric point-of-view
- We should get the same result as the single-threaded example
  - Right?

What could go wrong?

What did the programmer intend?

- In the third try, the OMP BARRIER shows clear programmer intent
  - The message from the thread where (tid.EQ.0) must “go first”
  - Presumed intent: the first message will not overtake the second

- In the first and second tries, there is no clear programmer intent
  - From the user’s point-of-view, either message could “go first”
  - Presumed intent: whatever is fastest/best for MPI?

What about thread support levels?

- Up to MPI_THREAD_SERIALIZED tells MPI there must be an order
  - In the first try, these thread support levels are erroneous
  - These levels mean MPI knows there is an execution order
    - Whether the programmer knows it or not
  - MPI will definitely “see” one MPI call before the other

- MPI_THREAD_MULTIPLE tells MPI nothing about order
  - MPI might “see” one MPI call before the other …
  - … or execution of the two MPI calls might be interleaved

How can MPI tell the difference?

- MPI cannot “see” the OpenMP BARRIER
  - Also, what about other forms of synchronisation?

- MPI cannot distinguish forced synchronisation from accidental timing
  - sleep(1000) or load imbalance look the same to MPI

- Insight: these three codes are ONE and the SAME example
  - (When using MPI_THREAD_MULTIPLE for all three)

What about thread compliance?

- Section 11.6.1 “1. All MPI calls are thread-safe, i.e., two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved.”

- For thread-compliant MPI implementations, even if the actual execution is interleaved (so, no defined relative execution order), there must be an “outcome order” (as if there was a relative execution order)

What about performance optimisations?

- Code to determine timing, which thread/call is “first”?
  - Clocks — system call, bad
  - Locks — reduced parallelism, ugly
  - Atomics — near-zero overhead, good enough?

- Code to transmit order/sequence — needed anyway
  - For example, single-threaded code but adaptively routed network

What about user-level threads?

- Can MPI reliably distinguish between different “threads”?
- Can MPI reliably identify which “thread” is calling into MPI?
- Can MPI reliably protect its internal data structures from “threads”?
- Does support for “threads” require integration with the (user-level) threading library/runtime?
- Does support for multiple types of “thread” require integration with multiple threading libraries/runtimes?
- Does this create a combinatorial build problem?

A little from camp A
A little from camp B

Camp A — “I will have order” (Dolores Umbridge)

- There is always a relative order for MPI calls
- Also, an ordering can always be imposed
- Parallel programming is hard enough as it is!
- When the order is clear, MPI must not disobey

Camp B — “Freedom!” (William Wallace)

- MPI cannot differentiate forced from accidental
- Determining programmer intent is impossible for MPI
- Imposing and/or enforcing order costs performance
- There are better ways to achieve order, e.g. tags

What can we do?

Tentative proposal(s)

1) Modify the definition of the overtaking rule to clearly state camp A/B
    a) There is no consensus on which camp to choose
    b) Both break something we don’t want to break <- bad

2) Create new thread support levels
    a) Camp A is between MPI_THREAD_SERIALIZED and MPI_THREAD_MULTIPLE
    b) Camp B is greater than MPI_THREAD_MULTIPLE
    c) MPI_THREAD_MULTIPLE is left alone — still ambiguous
3) New thread support levels are only permitted for sessions

16%

Questions?

What about the receiver-side?

1) We need a detailed treatment of the receive-side for the multithreaded sender (with single-threaded receiver) example codes.
    - Walk through all the ways to insert/remove items in the receiver's "queue"
    - Decide required/permitted/prohibited for each (with Standard references)
2) We need a detailed treatment of both sides of a multithreaded receiver (with single-threaded sender) example code.
    - Matching must be done by the receiver MPI process; does that change things significantly? If so, why and how?
3) We need a detailed treatment of both sides of a multithreaded sender & multithreaded receiver example code.

Which camp is right?

1) We need to lead the discussion away from “which camp is right?”

2) We should discover the pros and cons for choosing each approach.

3) We should discuss ways to offer both approaches in future.