

# MPI Finally Needs to Deal with Threads

Joseph Schuchart (Stony Brook University)

Joachim Jenke, Simon Schwitanski (RWTH Aachen)

EuroMPI/USA 2025, October 3, 2025, Charlotte, NC

---

# The Current State of Threads in MPI

'Logically concurrent' isn't #117

✓ Closed

 [mpi-forum/mpi-...#748 \(+1\)](#) ▾

400+ comments

**MPI\_THREAD\_SINGLE:** Only one thread will execute.

**MPI\_THREAD\_FUNNELED:** The process may be multithreaded, but the application must ensure that only the main thread makes MPI calls (for the definition of main thread, see [MPI\\_IS\\_THREAD\\_MAIN](#) on page 483).

**MPI\_THREAD\_SERIALIZED:** The process may be multithreaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from threads (all MPI calls are “serialized”).

**MPI\_THREAD\_MULTIPLE:** Multiple threads may call MPI, with no restrictions.

... is monotonic; i.e.,  $\text{MPI\_THREAD\_SINGLE} < \text{MPI\_THREAD\_FUNNELED} < \text{MPI\_THREAD\_SERIALIZED} < \text{MPI\_THREAD\_MULTIPLE}$ .

... in `MPI_COMM_WORLD` may require different levels of thread sup-

Change in thread safety in MPI 4.1 #846

⦿ Open

Threading guarantees of MPI\_User\_function #64

⦿ Open

Concurrent start and completion of persistent requests #858

⦿ Open



## Concurrent use of request in start[all] and completion functions

```

MPI_Recv_init(&bufa, 1, MPI_INT, 0, 42, MPI_COMM_SELF, req);
MPI_Irecv(&bufb, 1, MPI_INT, 0, 23, MPI_COMM_SELF, req + 1);
MPI_Wait(req, MPI_STATUS_IGNORE);
MPI_Test(req, &flag, MPI_STATUSES_IGNORE);
#pragma omp parallel sections
{
  #pragma omp section
  MPI_Waitall(2, req, MPI_STATUSES_IGNORE);
  MPI_Test(req, &flag, MPI_STATUSES_IGNORE);
}
MPI_Send(&bufc, 1, MPI_INT, 0, 42, MPI_COMM_SELF);
MPI_Wait(req, MPI_STATUSES_IGNORE);
MPI_Request_free(req);

```

Annotations in the diagram:

- `bufa = 0` (points to `bufa` in `MPI_Recv_init`)
- `true` (points to `flag` in `MPI_Test` inside the parallel sections)
- `bufa = 0` (points to `bufa` in `MPI_Irecv`)
- `bufa = bufc` (points to `bufc` in `MPI_Send`)
- `#pragma omp section` (points to the `MPI_Start` call in the second section)
- `usleep(10000);` (points to the `usleep` call in the second section)
- `MPI_Start(req);` (points to the `MPI_Start` call in the second section)
- `MPI_Send(&bufc, 1, MPI_INT, 0, 42, MPI_COMM_SELF);` (points to the `MPI_Send` call in the first section)

MPI\_Send cannot execute after MPI\_Waitall  
 ⇒ MPI\_Start executes before MPI\_Waitall?

→ A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

→ 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**.

# Thread-Compliant Implementations (§11.6.1)

## MPI 5.0 §11.6.1 (I)

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**.

## MPI 5.0 §11.6.1 (II)

**Blocking MPI calls will block the calling thread only**, allowing another thread to execute, if available. The calling thread will be blocked until the event on which it is waiting occurs. Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. **A blocked thread will not prevent progress of other runnable threads on the same process**, and will not prevent them from executing MPI calls.

## 'Logically concurrent' isn't #117

✓ Closed

[mpi-forum/mpi-...#748](#) (+1) ▼

*Advice to users.* The MPI Forum believes the following paragraph is ambiguous and may clarify the meaning in a future version of the MPI Standard. (*End of advice to users.*)

On the other hand, if the MPI 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.

*Advice to implementors.* The MPI Forum believes the previous paragraph is ambiguous and may clarify the meaning in a future version of the MPI Standard. (*End of advice to implementors.*)



# Conflicting Messages

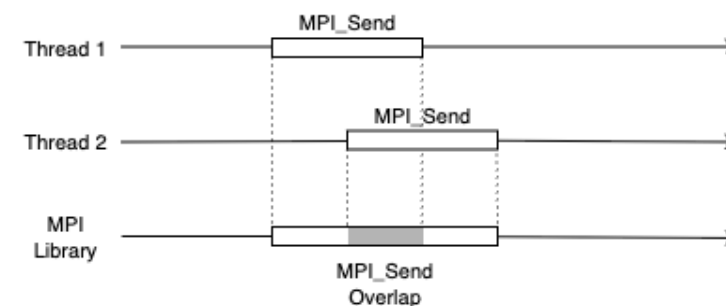
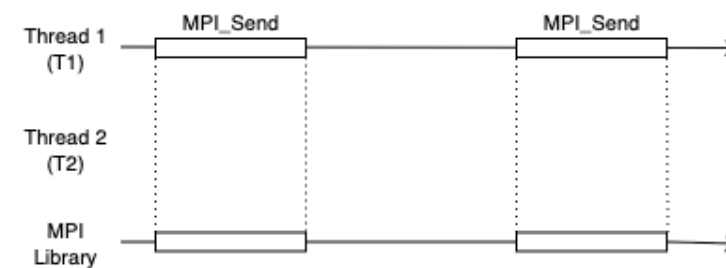
Same communicator, tag, and processes

## MPI 5.0 §3.5

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 **multi-threaded**, 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.



# A Tale of Two Interpretations (I)

## Stronger Interpretation

The application may not intentionally define an order. Messages must still be matched in the order they were posted.

Does  
"mpi\_assert\_allow\_overtake"  
only apply to single threads?

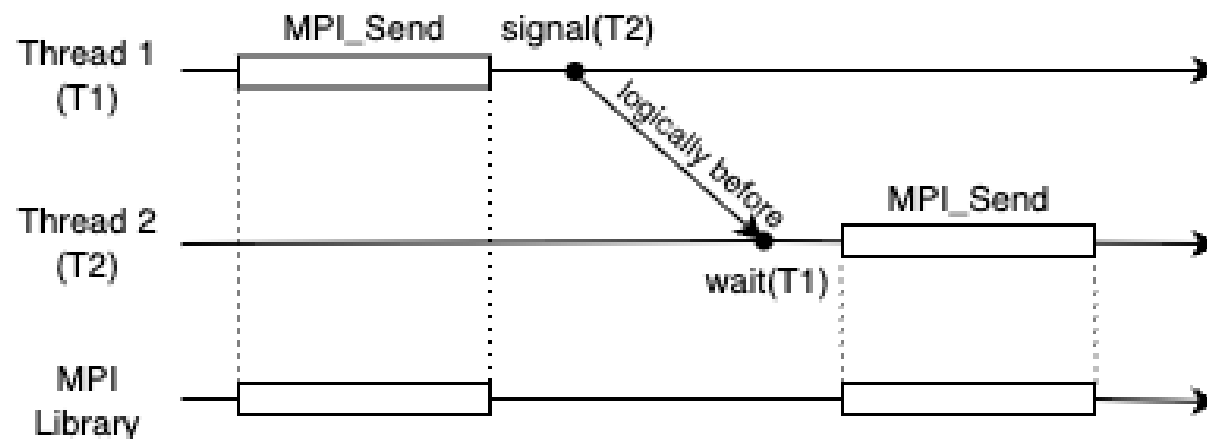
## Weaker Interpretation

MPI may treat **any** messages sent by two threads as logically concurrent. Their messages can be matched in any order.

### MPI 5.0 §11.6.1

It is the user's responsibility to prevent races when threads within the same application post conflicting communication calls. The user can make sure that two threads in the same process will not issue conflicting communication calls by using distinct communicators at each thread.

# Are These Sends “Logically Concurrent?”



No demonstrated benefit of assuming  
“Yes!”





## Principle of least astonishment

🌐 14 languages ▾

Article [Talk](#)

[Read](#) [Edit](#) [View history](#) [Tools](#) ▾

From Wikipedia, the free encyclopedia

*"Least surprise" redirects here. For the principle of least surprise in the Bayesian brain hypothesis, see [Free energy principle](#) and [Bayesian approaches to brain function](#).*

In [user interface design](#) and [software design](#),<sup>[1]</sup> the **principle of least astonishment (POLA)**, also known as **principle of least surprise (POLS)**,<sup>[a]</sup> proposes that a component of a system should behave in a way that most users will expect it to behave, and therefore not astonish or surprise users. The following is a corollary of the principle: "If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature."<sup>[4]</sup>

The principle has been in use in relation to computer interaction since at least the 1970s. Although first formalized in the field of computer technology, the principle can be applied broadly in other fields. For example, in [writing](#), a [cross-reference](#) to another part of the work or a [hyperlink](#) should be phrased in a way that accurately tells the reader what to expect.

### Origin [\[edit\]](#)

An early reference to the "Law of Least Astonishment" appeared in the [PL/I](#) Bulletin in 1967 (PL/I is a programming language released by IBM in 1966).<sup>[5]</sup> By the late 1960s, PL/I had become infamous for violating the law,<sup>[6]</sup> for example because, due to PL/I's precision conversion rules,<sup>[7]</sup> the expressions `25 + 1/3` and `1/3 + 25` would either produce a fatal error, or, if errors were suppressed, 5.3333333333 instead of the correct 25.3333333333.<sup>[8][9][10][11]</sup>

# What is Concurrency, Anyway?

## MPI 5.0 §3.5

[...] if the process is **multi-threaded**, 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**. [...]

Neglects  
synchronization  
outside of MPI

# What is Concurrency, Anyway?

## MPI 5.0 §3.5

[...] if the process is **multi-threaded**, 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**. [...]

Neglects  
synchronization  
outside of MPI

## Happens-Before Relation ( $\rightarrow$ )

1. If event A occurs before event B on the same parallel entity, then  $A \rightarrow B$ .
2. If a parallel entity in event A sends a signal to another parallel entity that blocks for it in an event B, then  $A \rightarrow B$ .
3. If  $A \rightarrow B$  and  $B \rightarrow C$ , then  $A \rightarrow B$ .

Operations without a  $\rightarrow$  relation are “logically concurrent.”

$\rightarrow$  can be determined through logical clocks

# MPI is Part of a Large System

As long as the application can guarantee a  
→ relation between two events, MPI  
should respect the **user's perceived order**.

## Proposal for §11.6.1

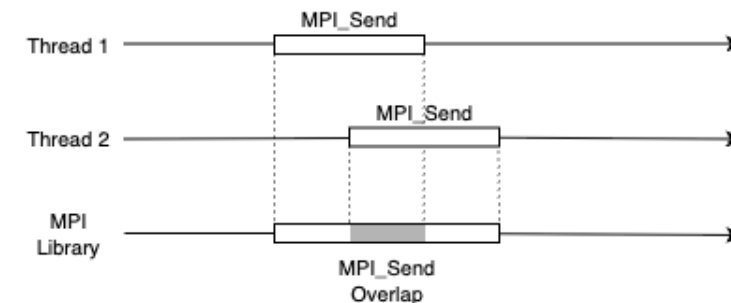
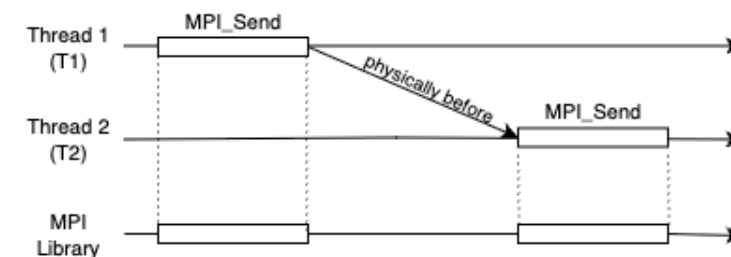
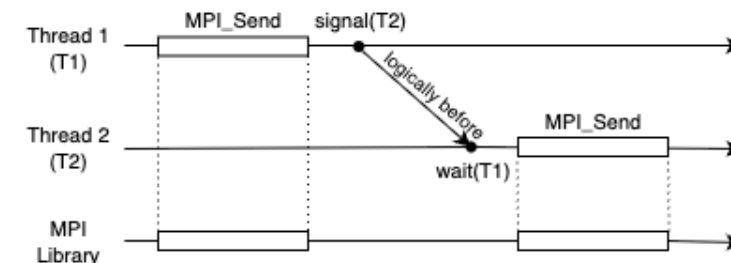
If a process has a single thread of execution, then any two communications executed by this process are ordered, i.e., they have an established happens-before relation.

On the other hand, if the process is multi-threaded, then two operations are considered **logically concurrent only if the application has not established a happens-before relation** (i.e., strict ordering) between the two messages. In such a case, the two messages sent can be received in any order. [...]

The same principle can be extended to MPI procedure calls: two MPI procedure calls are considered “logically concurrent” only if no happens-before relation between them has been established.

# Assumptions MPI Should Make

1. The application **has established** a happens-before relationship between two MPI procedure calls. The implementation must adhere to any ordering imposed by the application.
2. The application **may not have established** a happens-before relationship between two MPI procedure calls. This means that two calls to MPI may happen logically and even physically concurrent and the implementation must be able to choose an order.



# Conflicting Buffer Accesses

## MPI 5.0 §3.6

A **nonblocking send call** indicates that the system may start copying data out of the send buffer. The sender should not modify any part of the send buffer after a nonblocking send operation is called, until the send completes.

A **nonblocking receive call** indicates that the system may start writing data into the receive buffer. The receiver should not access any part of the receive buffer after a nonblocking receive operation is called, until the receive completes.

## MPI 5.0 §3.4

Does not  
mention  
READ &  
RECV.

Only applies  
to blocking  
operations!

In a multi-threaded implementation of MPI, the system may de-schedule a thread that is blocked on a send or receive operation, and schedule another thread for execution in the same address space. **In such a case it is the user's responsibility not to modify a communication buffer until the communication completes.** Otherwise, the outcome of the computation is undefined.



# Proposal: Conflicting Accesses

Provide clear guidance on what buffer accesses are conflicting and that their outcome is undefined.

## Proposal: Definition of Conflicting Access

Two logically concurrent memory accesses are conflicting if they access **overlapping memory regions** and at least one of them potentially **modifies the content** in that region.

For example, such conflicting accesses may occur if a thread reads the content of a buffer that is used by an incomplete receive operation or if a thread writes to a buffer that is used by an incomplete send operation.

Such conflicting accesses may occur with any MPI operation. The outcome of such behavior is undefined.

## Change in thread safety in MPI 4.1 #846

[🕒 Open](#)

Version 4.1 section 11.6.3 line 38

Multiple threads completing the same request. A program in which two threads block, waiting on the same request, is erroneous. Similarly, the same request cannot appear in the array of requests of two concurrent `MPI_{WAIT|TEST}{ANY|SOME|ALL}` calls. In MPI, a request can only be completed once. Any combination of wait or test that violates this rule is erroneous.

# Concurrent Request Completion

Concurrent  
MPI\_Test is  
valid?

## MPI 5.0 §11.6.2

A program in which two threads **block, waiting on the same request, is erroneous.**

Similarly, the same request cannot appear in the array of requests of two concurrent MPI {WAIT|TEST}{ANY|SOME|ALL} calls.

In MPI, **a request can only be completed once.** Any combination of wait or test that violates this rule is erroneous.

[...] be as if the calls **executed in some order**, even if their **execution is interleaved.**

One is allowed to call MPI\_TEST with a null or inactive request argument. In such a case the procedure returns with flag = true and empty status.

# Concurrent Request Completion

```
MPI_Request req;  
MPI_Isend(..., &req);
```

```
#pragma omp parallel shared(req)  
{  
    int flag;  
    while(!flag) MPI_Test(&req,..., &flag);  
}
```

Concurrent  
MPI\_Test is  
valid?

## MPI 5.0 §11.6.2

A program in which two threads **block, waiting on the same request, is erroneous.**

Similarly, the same request cannot appear in the array of requests of two concurrent MPI {WAIT|TEST}{ANY|SOME|ALL} calls.

In MPI, **a request can only be completed once.** Any combination of wait or test that violates this rule is erroneous.

[...] be as if the calls **executed in some order**, even if their **execution is interleaved.**

One is allowed to call MPI\_TEST with a null or inactive request argument. In such a case the procedure returns with flag = true and empty status.

# Concurrent Request Completion

```
MPI_Request req;  
MPI_Send_init(..., &req);  
MPI_Start(&req);  
#pragma omp parallel shared(req)  
{  
    int flag;  
    while(!flag) MPI_Test(&req,..., &flag);  
}
```

Concurrent  
MPI\_Test is  
valid?

## MPI 5.0 §11.6.2

A program in which two threads **block, waiting on the same request, is erroneous**.

Similarly, the same request cannot appear in the array of requests of two concurrent MPI {WAIT|TEST}{ANY|SOME|ALL} calls.

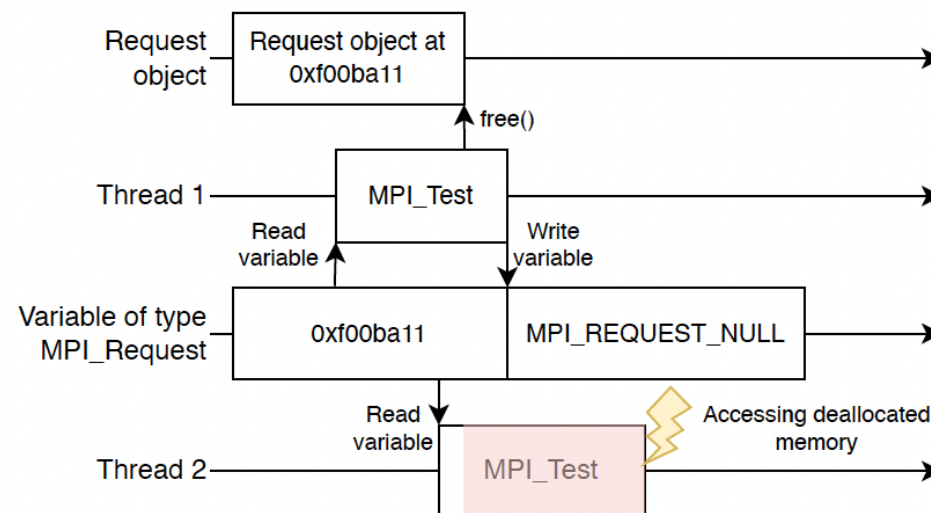
In MPI, **a request can only be completed once**. Any combination of wait or test that violates this rule is erroneous.

[...] be as if the calls **executed in some order**, even if their **execution is interleaved**.

One is allowed to call MPI\_TEST with a null or inactive request argument. In such a case the procedure returns with flag = true and empty status.

What about  
persistent  
requests?

# Allowing Concurrent Testing is a Bad Idea!

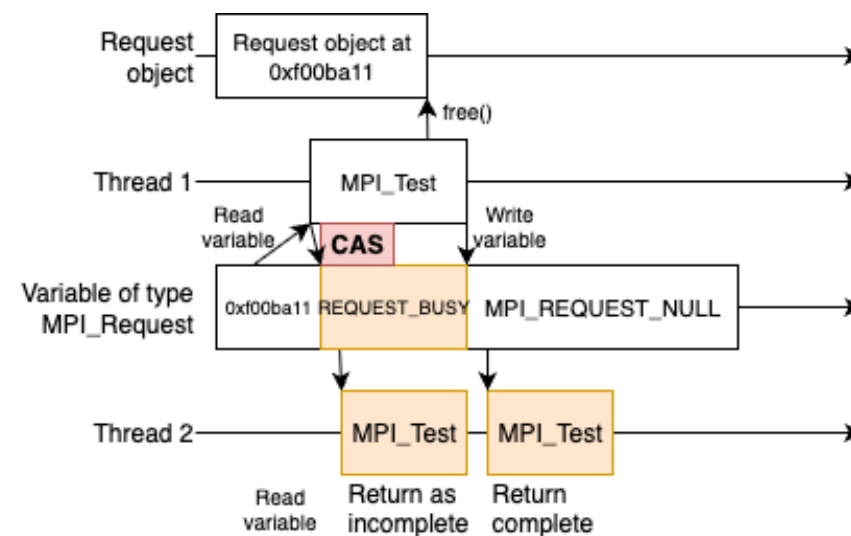


**Fig. 1.** Two threads attempting to test the same non-persistent request handle. Thread 1 completes the request and frees the request object. Thread 2 holds a stale handle to the freed request object because it did not see the completion of the request by Thread 1.



# Atomically Resetting the Handle

- CAS on every call to MPI\_Test
  - Thread takes ownership from variable holding the handle
  - Expensive to handle corner case
- New sentinel MPI\_REQUEST\_BUSY
  - To avoid reading potentially free'd handle
- Why allow concurrent MPI\_Test and not MPI\_Test[all|some|any]?



# Handle Equality: When are two MPI\_Request the same?

## MPI 5.0 §2.5.1

In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

## Proposal: Amended Handle Equality

In addition to their use by MPI calls for object access, handles can participate in assignments and comparisons.

**Two handles refer to the same MPI object if their type and value are identical.**

# Functions w/ Guaranteed Thread-Safety

## MPI 5.0 §11.6

Regardless of whether or not the MPI implementation is thread compliant, **a subset of MPI functions must always be thread-safe**. A complete list of such MPI functions is **given in Table 11.1**. When a thread is executing one of these routines, if another concurrently running thread also makes an MPI call, the outcome will be **as if the calls executed in some order**.

Table 11.1: List of MPI Functions that can be called at any time within an MPI program, including prior to MPI initialization and following MPI finalization

MPI_INITIALIZED
MPI_FINALIZED
MPI_GET_VERSION
MPI_GET_LIBRARY_VERSION
MPI_ABI_GET_VERSION
MPI_ABI_GET_INFO
MPI_INFO_CREATE
MPI_INFO_CREATE_ENV
MPI_INFO_SET
MPI_INFO_DELETE
MPI_INFO_GET_STRING

## MPICH Info page for MPI\_Info\_set

The MPI standard defined a thread-safe interface but this does not mean that all routines may be called without any thread locks. **For example, two threads must not attempt to change the contents of the same MPI\_Info object concurrently**. The user is responsible in this case for using some mechanism, such as thread locks, to ensure that only one thread at a time makes use of this routine.

# A Tale of Two Interpretations (II)

## Open MPI (Wide)

All conflicting accesses to MPI objects are protected by MPI\_THREAD\_MULTIPLE

## MPICH (Narrow)

Only non-conflicting accesses are protected by MPI\_THREAD\_MULTIPLE

# Which Parts of MPI Should be Thread-Safe?

Global State (session state)

MPI Objects (communicators, requests, info...)

MPI Handles (MPI\_Request, MPI\_Comm)

# Which Parts of MPI Should be Thread-Safe?

Global State (session state)

MPI Objects (communicators, requests, info...)

MPI Handles (MPI\_Request, MPI\_Comm)



### MPI 5.0 §11.6.2

A program in which two threads **block, waiting on the same request, is erroneous.**

Similarly, the same request must not appear in the array of requests of two concurrent MPI {WAIT|TEST}{ANY|SOME|ALL} calls.

In MPI, a **request can only be completed once.** Any combination of wait or test that violates this rule is erroneous.

# Wide: Disallow Concurrent Release

- Allows for conflicting accesses to info objects.
- Would allow concurrent completion of **persistent** requests.
- Would **not** permit concurrent Wait/Start.

## Proposal: §11.6.2

A program in which two threads concurrently pass the same MPI handle to MPI procedures and one of them may release the MPI object (i.e., replace the MPI handle value) is erroneous. For example, a non-persistent request can only be completed once and a communicator can only be freed once.

# Narrow: Allow Concurrent Modification of Non-User Observable State

- Would prohibit concurrently setting info keys on MPI objects
- Disallows concurrent test/wait/start since completion is user-visible!
- Still allows concurrent P2P & RMA operations

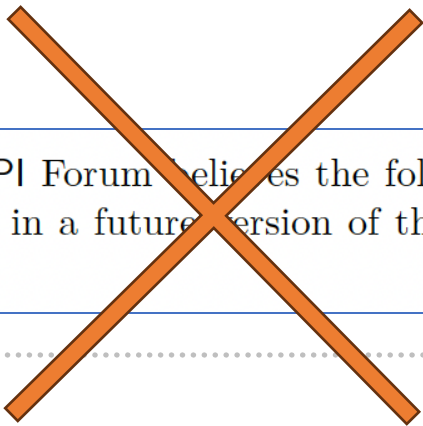
The community should choose the level of concurrency supported by MPI.

## Proposal: §11.6.2

A program in which two threads concurrently pass the same MPI handle to MPI procedures and one of them may alter the **user-observable state** of the MPI object or the handle itself is erroneous. For example, a request can only be completed once and a communicator can only be freed once.

# Conclusions

- Details of thread-safety in the standard are unclear.
- Identified inconsistencies in the standard and in implementations interpreting the standard
- Provided suggestions for improvements.



*Advice to users.* The MPI Forum believes the following paragraph is ambiguous and may clarify the meaning in a future version of the MPI Standard. (*End of advice to users.*)