MPI Windows model

1. What an MPI window *is* (public vs private)
2. The two memory models: **unified** vs **separate**
3. Epochs and synchronization (fence, PSCW, lock, flush, sync)
4. Interaction with local loads/stores (especially for shared-memory windows)
5. Atomicity and overlapping accesses
6. Practical patterns & common traps

{ref [1]}Checklist

---

1. Windows: the “two copies”

{ref [2]}

In MPI RMA, each process exposes a region of its memory as a **window** (MPI_Win). Other processes can then issue RMA operations (Put/Get/Accumulate/Get_accumulate) into that region.

Conceptually, MPI distinguishes:

- **Private window copy** – what your process sees via normal loads/stores (and sends).
- **Public window copy** – what RMA operations conceptually access.

That “two copies” model is *conceptual*; it lets MPI cover both cache-coherent shared memory and bizarre non-coherent NIC/device memory under one same picture.

The actual memory model is then defined in terms of how these copies relate, and how they synchronize.

---

2. Unified vs Separate memory models

{ref [3]}

For *each* window, MPI assigns a memory model, exposed via the window attribute `MPI_WIN_MODEL` (value is `MPI_WIN_UNIFIED` or `MPI_WIN_SEPARATE`).

You can query it with:

```
int model;
int flag;
MPI_Win_get_attr(win, MPI_WIN_MODEL, &model, &flag);

if ( flag && (model == MPI_WIN_UNIFIED) ) {
... }
else if ( flag && (model == MPI_WIN_SEPARATE) ) {
... }
```

2.1 Unified memory model

Public and private copies are *logically identical*.

The unified model, introduced in MPI-3, assumes that public and private copies are identical and requires hardware cache coherence to function correctly. This model is conceptually simpler and aligns with how modern shared-memory systems actually work.

- A store to window memory by the local process becomes **eventually** visible to remote Get/Accumulate operations, without extra MPI calls.
- A Put/Accumulate to the window becomes **eventually** visible to local loads without extra MPI calls. (mpi-forum.org)

“Eventually” is doing a lot of work: the standard doesn’t promise any instant visibility; it just guarantees that after appropriate synchronization (flushes, epoch ends, etc.) the view is consistent. Within an epoch and without flushes, you can absolutely see “torn” or intermediate values.

A key implication of the unified model is that it allows concurrent local and remote access to the same window locations, which would be problematic in the separate model. This flexibility comes at the cost of requiring hardware support for memory coherence
This is the model you get on normal cache-coherent CPU memory in most implementations. Shared-memory windows (`MPI_Win_allocate_shared`) *assume* unified semantics in practice; MPI even says it doesn’t define shared-window behavior under the separate model. (mpi-forum.org).

The memory model is not selectable by the programmer as part of the window creation call itself.
Instead, the memory model is determined by the MPI implementation and the underlying hardware architecture.
The *Unified model* is used by most modern MPI implementations if the system provides hardware cache coherence (as in most shared-memory systems today).

The *Separate model* is used only on platforms that lack memory coherence, or if the MPI library requires software-level consistency. This is rare on modern clusters. However, on some *aarch64* that may be the case.

2.2 Separate memory model

Public and private copies are *logically separated*.

The separate model, inherited from MPI-2, is designed for maximum portability and maintains a conceptual distinction between public and private copies of window data. In this model, each MPI process maintains two logical views of memory:

- **Local load/store** → private copy; the original variables instances in process memory, accessed by local loads and stores as well as regular MPI operations like send/receive.
- **RMA operations** → public copy; a distinct copy of each variable for each window containing it, accessed by remote operations (Put, Get, Accumulate).

The fundamental constraint of the separate model is that it provides software coherence rather than relying on hardware cache coherency.
Synchronization calls (epoch boundaries, `MPI_Win_sync`) are responsible for moving data between public and private copies.

So, roughly:

- After remote Puts into the public copy, you **must** do MPI-level sync to see them via local loads.
- After local stores into the private copy, you must sync to make those stores visible to remote RMA.

Separate gives maximal portability (think exotic non-coherent networks, device memory), but at the cost of more explicit synchronization.

---

3. Epochs and synchronization

RMA operations must occur inside **epochs**, and synchronization calls define both:

- *who is allowed to do RMA to whom* (access vs exposure epochs)
- and *what ordering / completion guarantees* you get.

There are three families of synchronization:

1. **Fence (collective on the window)**
2. **PSCW (post/start/complete/wait) – active target, non-collective over subgroups**
3. **Lock/lock_all (passive target)**

Plus the fine-grained completion operations:

- `MPI_Win_flush`, `MPI_Win_flush_local`, `MPI_Win_flush_all`, `MPI_Win_flush_lcalo_all` → Forces completion of RMA operations **from origin to target**.
- `MPI_Win_sync` (public/private synchronization) → Synchronizes **local memory** with the window (target-side operation)
- `MPI_Win_unlock(_all)` / end of fence / PSCW act as “bulk” completions

3.1 Access vs exposure epochs

- **Access epoch** on an origin: period during which it may issue RMA ops to some target(s) on a given window.
- **Exposure epoch** on a target: period during which it may be the target of RMA ops on that window.

Each sync mechanism opens/closes these epochs in specific patterns.

SEE "One_sided_data_transfer.pdd"

3.2 Flush and sync

Key ones:

- `MPI_Win_flush(target, win)`
  All outstanding RMA ops from this origin to `target` on `win` are complete at both origin and target when it returns. (Remote completion.) (mpi-forum.org)
- `MPI_Win_flush_local(target, win)MPI_Win_lock`
  Only guarantees local completion (origin buffer reusable), not that the target has seen the data.
- `MPI_Win_sync(win)`
  Synchronizes public and private copies on *the calling process*, conceptually like closing and reopening

epochs for the purpose of public/private consistency, but it does **not** complete pending RMA itself. (mpi-forum.org)

This is the RMA equivalent of a memory fence between local loads/stores and RMA traffic *on the same process*.

---

4. Local loads/stores vs RMA: happens-before intuition

As in OpenMP, there is a relation `hb` ("happens-before") defined by:

- program order on each process, plus
- synchronization edges from:
  - epoch boundaries (fence, PSCW)
  - lock/unlock
  - flush / flush_local (for RMA completion)
  - win_sync (for public/private consistency on a process)

MPI's prose version of `hb` basically says:

- RMA effects become visible to a target's **public** copy at remote completion.
- In unified model, that becomes visible to local loads *eventually* and definitely by the right synchronization points.
- In separate model, that becomes visible to local loads only after a `Win_sync`/epoch boundary that pushes public→private.

Note:

- `MPI_Barrier` gives **process synchronization but not memory synchronization**; it does *not* create RMA-public/private ordering edges. The standard is explicit: "MPI_BARRIER provides process synchronization, but not memory synchronization." (mpi-forum.org)

So any pattern that relies on "I did Puts, then `MPI_Barrier`, so everyone must see my new values via local loads" is **not** portable. You need RMA synchronization, not just a collective.

4.1 Example: simple Put + read pattern (unified model)

Origin P:

```
MPI_Win_lock(MPI_LOCK_SHARED, Q, 0, win);
MPI_Put(buf, n, MPI_DOUBLE, Q, disp, n, MPI_DOUBLE, win);
MPI_Win_flush(Q, win);   // ensure remote completion
MPI_Win_unlock(Q, win);
```

Target Q (reading via local loads):

- In **unified** model, after the flush/unlock completes and appropriate synchronization on Q (another RMA sync or `MPI_Win_sync`), Q can safely read the new values via plain loads.
- A *bare* `MPI_Barrier` is not guaranteed to make caches coherent, though most real systems "accidentally" do.

For portable correctness when Q uses local loads:

```
// Q wants to read values that P put
MPI_Win_lock_all(0, win);   // optional depending on scheme
MPI_Win_sync(win);          // pull public -> private (unified or separate)
read_from_window();         // plain loads
MPI_Win_unlock_all(win);
```

On typical CCNUMA CPUs with unified model, many implementations ensure that `flush` + `unlock` + `win_sync` act like a full memory fence with respect to the window.

4.2 Separate model example

Same code, but separate model:

- P's Put updates **public** copy.
- Q's plain loads read **private** copy.
- Without `MPI_Win_sync(win)` on Q, Q could keep seeing stale data indefinitely.

So for portable code that might run on separate:MPI_Win_lock

```
// P: as before, Put + flush / unlock

// Q:
MPI_Win_lock_all(0, win);
MPI_Win_sync(win);          // public -> private
read_from_window();
MPI_Win_unlock_all(win);
```

That is the crucial difference: in unified you get eventual consistency "for free" but still need ordering; in separate, you must explicitly move data between copies.

Well.. in principle either you explicitly check the model or you use the safest (and more costly) pattern for separate model.

---

5. Atomicity and overlapping accesses

MPI is also quite picky about **concurrent accesses to the same location** in a window.

5.1 Compatibility of operations

There's a compatibility matrix in tutorials and derived from the spec: for each pair of operations (load/store/Get/Put/Accumulate) accessing the same target location concurrently, it says whether overlapping accesses are allowed and whether the result is defined, undefined, or just "garbage but legal." (wgropp.cs.illinois.edu)

High-level gist:

- Many combinations require **non-overlapping** regions to be well-defined.
- Concurrent RMA writes (Put/Accumulate) to the same location from different origins are generally **not**

ordered or atomic, except for special cases involving Accumulate.

5.2 Accumulate and info keys

For Accumulate-style operations (Accumulate, Get_accumulate, Fetch_and_op, etc.), MPI provides *per-element atomicity* and some ordering guarantees, but with caveats:

- For a single origin → target, overlapping Accumulate operations to the same location are **strictly ordered** by default: they commit in program order. (mpi-forum.org)
- Ordering is controlled by the info key `accumulate_ordering` on window creation:
  - Default: `"rar,raw,war,waw"` → all the usual read/write-ordering relations enforced.
  - You can weaken this by specifying a subset or `"none"` to let the implementation reorder for performance. (mpi-forum.org)
- Info key `accumulate_ops` (default `same_op_no_op`) constrains *what* concurrent accumulate operations are allowed on the same location (e.g. they must use same op or `MPI_NO_OP`). (GitHub)

For correctness:

- If multiple origins concurrently accumulate to the same location, you only get well-defined results when they obey these constraints. Otherwise, it's erroneous.

Note also: the atomicity is per *datatype instance* (per element), not necessarily per__threadfence_system byte.

---

6. Shared-memory windows and direct loads/stores

For `MPI_Win_allocate_shared`, all processes in a node share a piece of memory directly addressable via `baseptr` and `MPI_Win_shared_query`. (cvw.cac.cornell.edu)

Important subtleties:

- MPI only defines consistent load/store semantics for shared-memory windows in the **unified** model, using standard RMA synchronization (fence, lock/unlock, flush, `MPI_Win_sync`). (mpi-forum.org)
- You're allowed to:
  - Do plain `*ptr = value;` writes and reads on that shared region, and
  - Use RMA operations to the same region.
- But you **must** use MPI's RMA sync functions to define ordering and visibility. `MPI_Barrier` alone is not sufficient. (Stack Overflow)

Typical safe pattern for shared-memory windows:

Writer (proc P):

```
// local store
shm[idx] = new_value;
/* ensure hardware/compiler does not reorder too wildly */
MPI_Win_sync(win);     // sync private -> public
MPI_Barrier(comm_shm); // process sync (optional but practical)
```

Reader (proc Q):

```
MPI_Barrier(comm_shm); // wait until writer reached sync point
MPI_Win_sync(win);     // public -> private
val = shm[idx];
```

In practice, we often rely on the unified model and hardware coherence, but the strictly portable semantics are as above.

---

7. Common misconceptions and gotchas

Let's play bad-ideas bingo and then fix them:

1. **"A window is just shared memory, so I can treat it like OpenMP."**
   Not quite. A window is memory with *global* visibility, but consistency is governed by MPI's epochs and memory model, not OpenMP's. You must think in terms of public/private and RMA synchronization, especially for overlapping RMA and local accesses.
2. **"MPI_Barrier is a memory fence."**
   No. The spec explicitly says it is *only* a process sMPI_Win_lockynchronization, not a memory synchronization. (mpi-forum.org)
3. **"Unified model means I don't need `MPI_Win_sync`."**
   Not always. Unified means eventual coherence *exists*, not that ordering is magically what you want. When you mix fine-grained local loads/stores with RMA, you often still need `Win_sync` and/or epoch boundaries for defined behavior.
4. **"If it works on my x86 cluster, it's standard-compliant."**
   x86 TSO + cache coherence is much stronger than MPI requires. Many "works on my laptop" patterns would be invalid on weaker architectures or device memory. The separate model is essentially the spec's way of telling you: *don't count on accidental coherence*.
5. **"Concurrent Puts/Accumulates to the same location are always fine; last writer wins."**
   Only certain patterns are legal, and accumulate has special constraints. If multiple origins modify the same location without respecting the compatibility/ordering rules, the behavior is erroneous, not "unspecified but benign".

---

8. Take-away summary

If we compress the whole mess into a few rules of thumb:

- Think of each window location as having:
  - a **public** version (for RMA), and
  - a **private** version (for local loads/stores).
- **Unified model:** these are physically the same object, but still governed by MPI's synchronization rules for visibility/order.
- **Separate model:** these are logically distinct; you must copy between them via epoch boundaries / `MPI_Win_sync`.

For correctness:

- Only do RMA between proper access/exposure epochs.

- Use `MPI_Win_flush(_local)`/unlock/fence/PSCW to define when RMA effects are complete.
- Use `MPI_Win_sync` when you mix RMA with local loads/stores on the same window.
- Never rely on `MPI_Barrier` as a memory fence.
- For overlapping updates, use Accumulate with appropriate info keys, or avoid concurrency on individual locations.