One-Sided Communication in MPI: comments about real performance

Basic concepts

MPI Remote Memory Access (RMA) operations fundamentally break from the two-sided paradigm by decoupling data transfer from synchronization. However, this conceptual simplicity masks significant complexity in the memory consistency model (see notesd on memory model).

Window Creation Strategies

MPI_Win_create: Maps existing memory into the window

```
MPI_Win_create(void *base, MPI_Aint size, int disp_unit,
               MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

Critical: The `disp_unit` parameter is often misunderstood. Setting it to anything other than 1 (byte) can cause subtle bugs when mixing datatypes. The displacement calculation becomes:

```
target_addr = base + (target_disp × disp_unit)
```

MPI_Win_allocate: Allocates and maps memory atomically

```
MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
                 MPI_Comm comm, void *baseptr, MPI_Win *win)
```

Reality check: While MPI_Win_allocate promises better optimization opportunities, actual performance benefits are highly implementation-dependent. Many implementations simply wrap malloc() without exploiting special memory registration.

Communication Operations

The three fundamental operations exhibit different consistency guarantees:

MPI_Put: `origin → target`
MPI_Get: `target → origin`
MPI_Accumulate: `origin ⊕ target → target`

Critical: These operations are *not* atomic by default. Without proper synchronization, concurrent Put operations to overlapping memory regions yield undefined behavior.

Synchronization Modes: The Devil in the Details

1. Fence Synchronization

```
MPI_Win_fence(int assert, MPI_Win win)
```

The fence model implements a BSP-like pattern with collective synchronization. The mathematical model:

Let E_i be the epoch at process i. A fence creates a barrier such that:

- All RMA operations in epoch $E_i^{(n)}$ complete before any operation in $E_i^{(n+1)}$ begins
- $\forall i, j : E_i^{(n)} \rightarrow E_j^{(n+1)}$ (happens-before relationship)

Critical: Fence synchronization is collective even when communication is sparse. For a communication pattern with degree $d \ll p$, you still pay $O(\log p)$ synchronization cost (d is the “density” of the communications and p is the number of processes).

2. PSCW (Post-Start-Complete-Wait)

```
MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
MPI_Win_complete(MPI_Win win)
MPI_Win_wait(MPI_Win win)
```

Critical complexity: Group management overhead often negates performance benefits. Creating and freeing MPI_Groups for dynamic communication patterns can introduce significant overhead. Best when you have static recurrent pattern, for which you can built the groups only once.

3. Lock/Unlock (Passive Target)

```
MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
MPI_Win_unlock(int rank, MPI_Win win)
```

Shared vs Exclusive locks:

- SHARED: Multiple processes can concurrently Get
- EXCLUSIVE: Single process has Put/Get/Accumulate access

Critical (misconception): despite the unfortunate naming, MPI locks are *not* mutex-like. They don't provide mutual exclusion for the target process's local access. The target can modify its window memory during a locked epoch, leading to race conditions.

4. Lock_all - The Scalability Trap

```
MPI_Win_lock_all(int assert, MPI_Win win)
MPI_Win_unlock_all(MPI_Win win)
```

Hidden cost: Implementations may need to track $O(p)$ state even for sparse communication. This seemingly convenient API can destroy scalability.

Memory Consistency: Theory Meets Reality

MPI defines two consistency models:

Separate Model: Public and private copies may diverge during epochs
Unified Model: Single copy visible to all processes

The unified model requires cache-coherent hardware or software emulation. You can query:

```
MPI_Win_get_attr(win, MPI_WIN_MODEL, &memory_model, &flag)
```

Critical issue: On non-cache-coherent systems (many HPC clusters), the separate model requires explicit synchronization even for local accesses:

```
MPI_Win_sync(win)  // Make remote changes visible locally
```

Writing a portable code (i.e. that works in every case) may be confusing and not that easy.
A nowadays important case is the usage of GPU, which is still very complicated, at least because there are two very distinct cases: the PCIe one, that falls in the category separate model, and the emerging on-chip (for instance: Grace-Hopper).

Performance Analysis and Common Pitfalls

Latency Hiding Myth

Common claim: "RMA hides latency by overlapping communication and computation"

Reality check:

1. MPI_Put/Get are often *synchronous* at small message sizes
2. Progress requires MPI_Win_flush or synchronization calls
3. True asynchronous progress needs:
    - Hardware RDMA support
    - Dedicated progress threads (which brings overhead)
    - Or frequent MPI calls (MPI_Iprobe, etc.)

Bandwidth Utilization

For large messages, theoretical bandwidth B is reduced by:

- Memory registration overhead: $O(n)$ for first touch
  a good practice is not to create and destroy windows: decide at the beginning and keep them resident
- Completion notification latency: L_{notify}
- Progress engine polling frequency: f_{poll}

Effective bandwidth: $B_{eff} = B \cdot \frac{n}{n+t_{reg}} \cdot \frac{1}{1+L_{notify} \cdot f_{poll}}$

Atomicity Guarantees

MPI_Accumulate atomicity: Only guaranteed for:

- Predefined operations (MPI_SUM, etc.)
- Same-sized, aligned accesses
- Within a single window

Counter-example: This is *not* atomic:

```
MPI_Accumulate(&val1, 1, MPI_INT, target, 0, 1, MPI_INT, MPI_SUM, win)
MPI_Accumulate(&val2, 1, MPI_INT, target, 0, 1, MPI_INT, MPI_SUM, win)
// These may interleave with other processes' operations!
```

When Should You Actually Use RMA?

Legitimate use cases:

1. Dynamic sparse data structures (adaptive mesh refinement)
2. Task pools with work stealing
3. Global hash tables or distributed caches
4. PGAS language implementations

Questionable use cases:

1. Regular stencil computations (use neighborhood collectives)
2. Dense linear algebra (use collective operations)
3. Simple producer-consumer (use point-to-point)

Implementation-Specific Realities

Intel MPI: Prefers MPI_Win_allocate with specific Info hints:

```
MPI_Info_set(info, "alloc_shm", "true")  // Shared memory optimization
```

Open MPI: Performance heavily depends on BTL selection:

- `vader` BTL: Shared memory via XPMEM or CMA
- `ucx` BTL: Better RMA but higher memory footprint

Cray MPI: Native uGNI provides true one-sided RDMA but requires:

```
MPI_Info_set(info, "cray_symmetric_heap", "true")
```

Critical Assessment

An often critical contradiction: MPI RMA promises simplicity through one-sided semantics but delivers complexity through its memory model and synchronization requirements.

The reality on performance: In most real applications, well-tuned two-sided communication outperforms RMA because:

1. Better MPI implementation optimization effort
2. Clearer semantics leading to fewer bugs
3. More predictable performance characteristics

A paradox: To ensure correctness, RMA often requires as much synchronization as two-sided communication, negating its primary advantage.