CONNECTING THE DOTS

ISC High Performance

JUNE 22–26, 2026 | HAMBURG, GERMANY

BITS Pilani
K K Birla Goa Campus

Analysis of Asynchronous I/O Interfaces for MPI Applications on NVMe Storage

Kaaviya Uthirapandian*, Druva Dhakshinamoorthy*, Riddhiman Sengupta, and **Arnab K. Paul**

*Kaaviya U. and Druva D. are both lead authors in this work and contributed equally.

Introduction

Today's MPI Usage

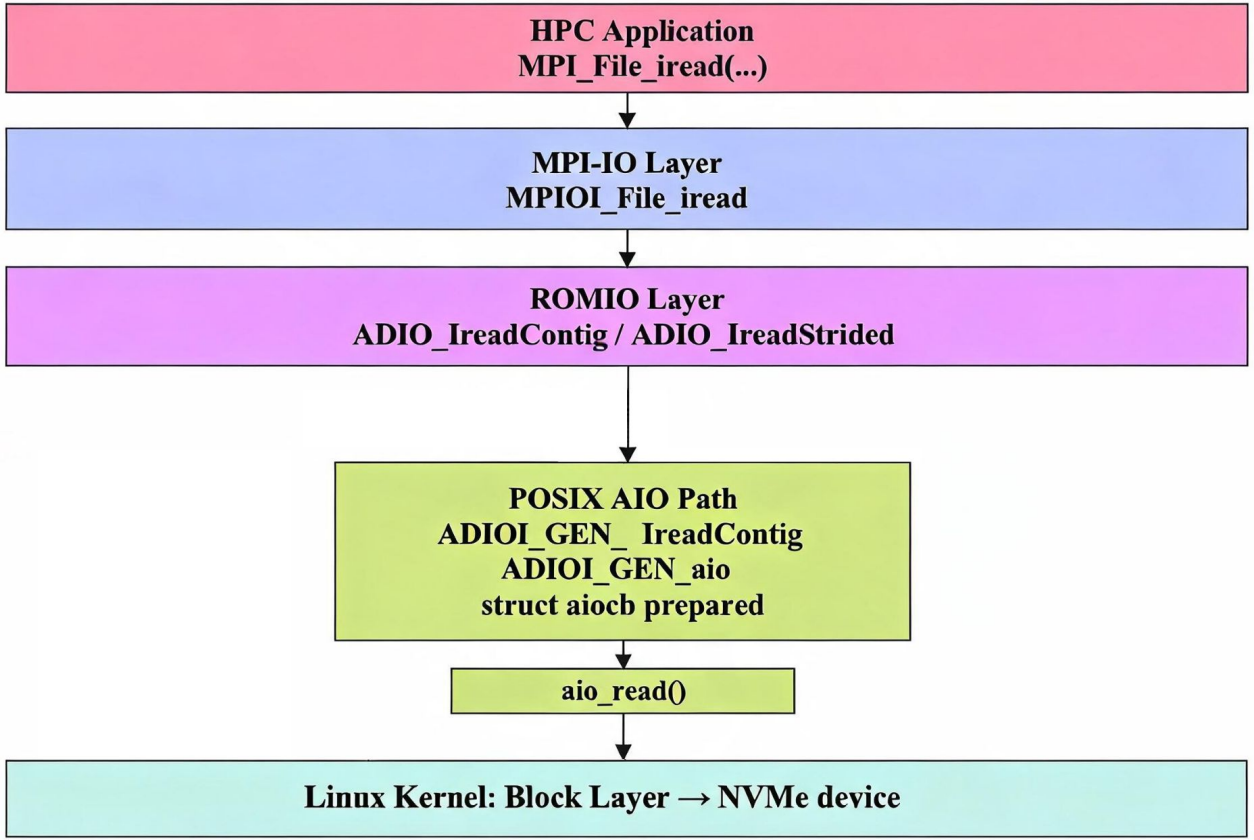
- Most production HPC codes still use blocking, per-rank MPI-IO.
- Typical pattern: `MPI_File_read_at` → stall until completion → next request.

Modern NVMe hardware

- Blocking, one-op-at-a-time I/O was reasonable for devices that could only handle a single request.
- Modern NVMe SSDs are fundamentally parallel: deep queues, peak throughput only at high in-flight request counts.

Current Implementation of Asynchronous I/O in MPICH

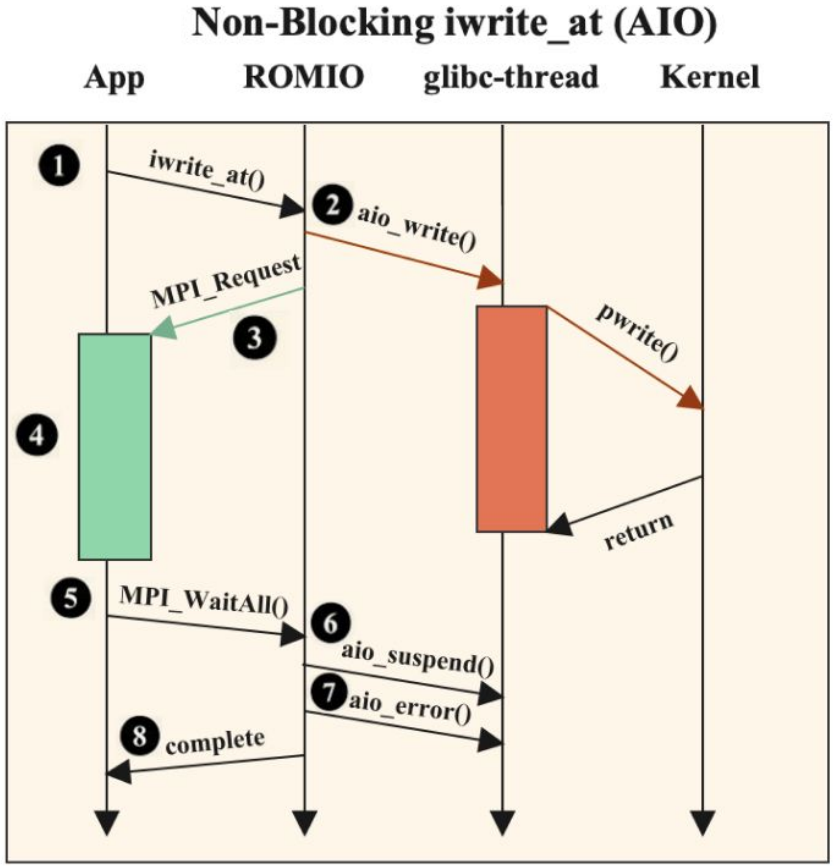
- MPICH's ROMIO layer already implements non-blocking MPI-IO using POSIX AIO as the asynchronous backend.
- In principle, this should let MPI-IO exploit device parallelism



POSIX AIO: Designed for slow storage

- **Helper-thread model**: each AIO request spawns a userspace helper thread that issues a blocking pread/pwrite. More concurrency → more threads to schedule
- **Per-operation syscall**: each `aio_read`/`aio_write` is its own syscall. No batching of submissions.
- **O(N²) completion path**: the application calls `aio_suspend`, scans all outstanding calls with `aio_error`, doing an O(N) scan per wakeup and O(N²) work in the worst case for N operations.

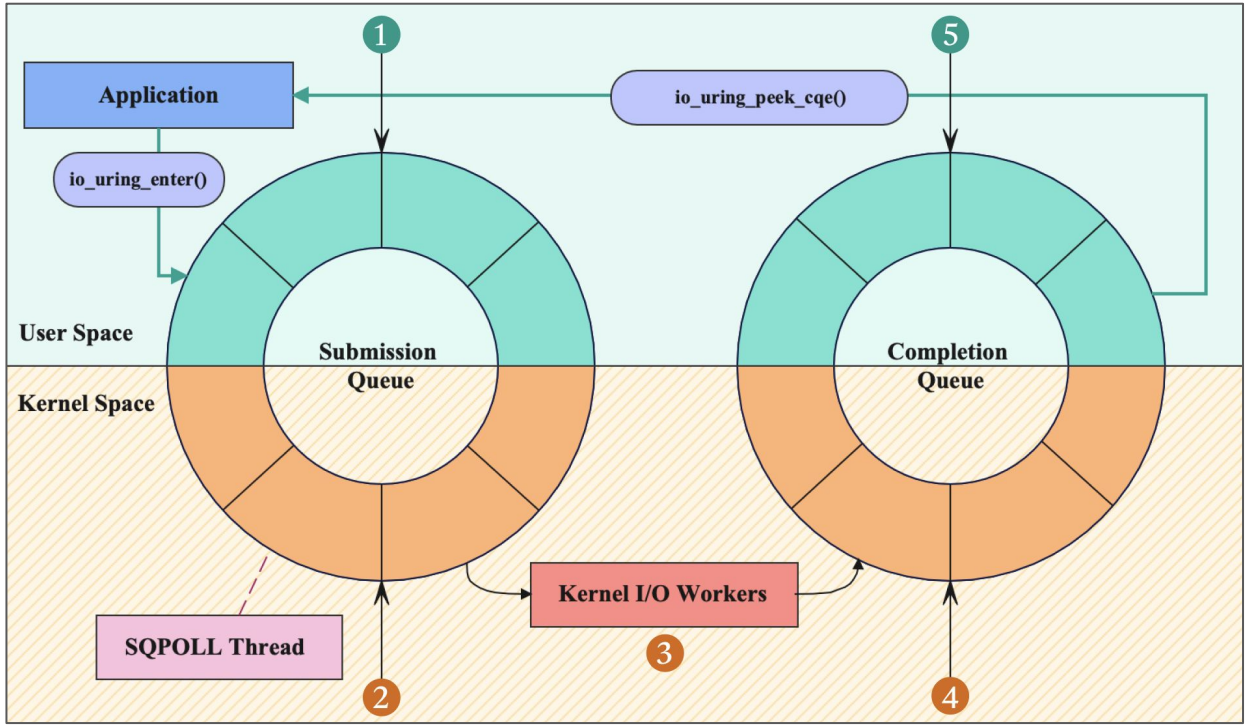
Together, these behaviors make POSIX AIO's software path increasingly expensive as concurrency rises, penalizing exactly the bursty, high-concurrency workloads that should benefit from asynchronous I/O.



io_uring: Async I/O aligned with NVMe

- **Shared-memory SQ/CQ rings:** Applications enqueue many submission queue entries (SQEs) into a shared-memory SQ ring and notify the kernel with a single `io_uring_enter()` call, amortizing syscall overhead across the batch.

- **Kernel I/O worker threads, no userspace helper threads:** Kernel I/O worker threads process SQEs directly at the block layer; the application thread does not spawn per-request helper threads in userspace.



io_uring: Async I/O aligned with NVMe

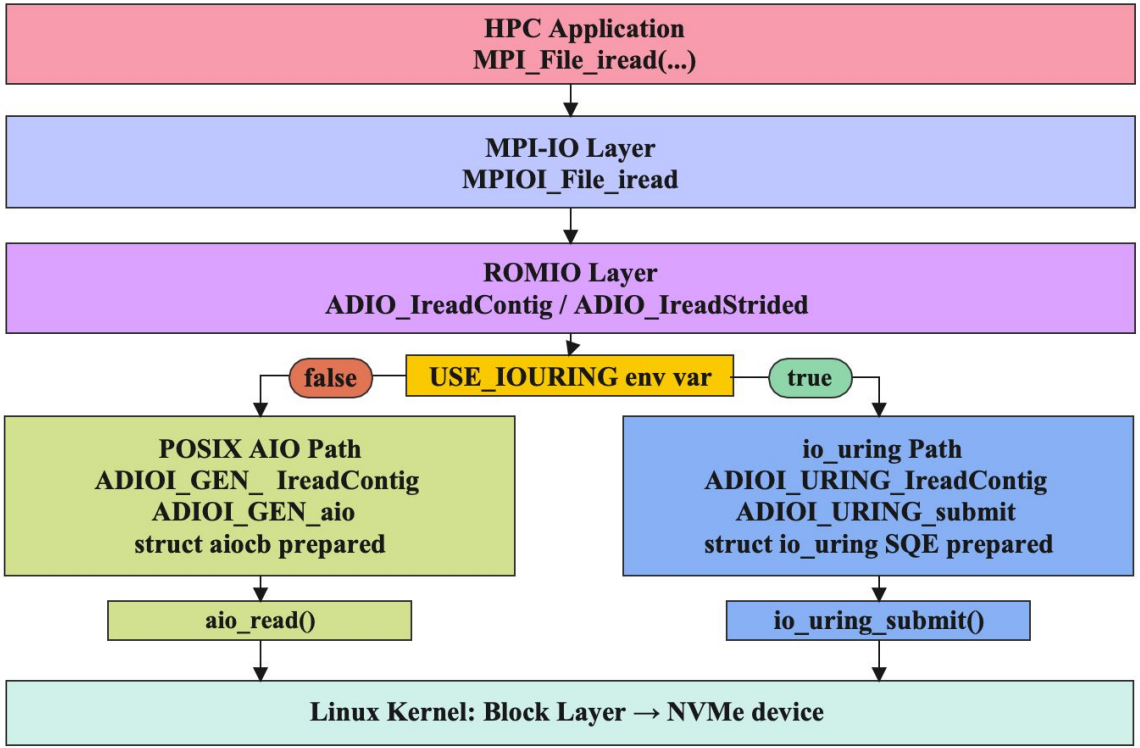
O(N) tagged completion via CQ ring

- On completion, the kernel writes one completion queue entry (CQE) per finished I/O into the CQ ring; each CQE carries a user-defined tag (`user_data`) set at submission time.
- The application drains the CQ ring by advancing a head pointer in shared memory, doing O(1) work per completed operation and O(N) total for N operations, instead of POSIX AIO's O(N²) `aio_suspend` + `aio_error` scanning.

io_uring replaces per-op syscalls and helper threads with batched submissions and O(N) tagged completions, reducing software-path overhead exactly in the high-concurrency regime where NVMe's internal parallelism matters.

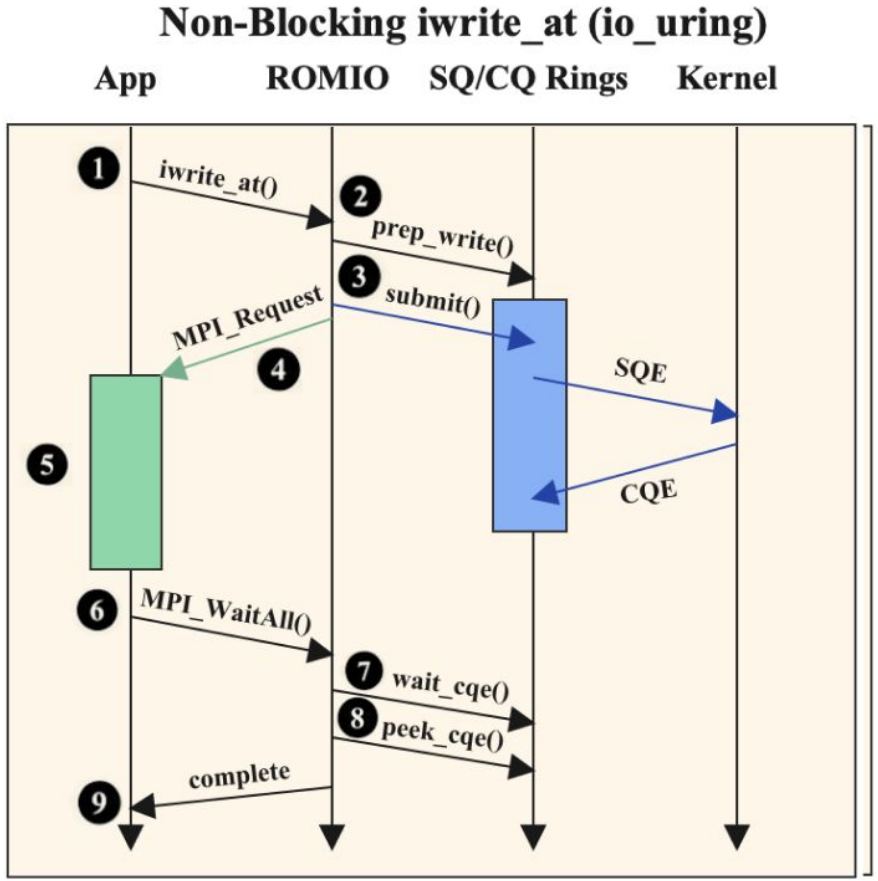
Integration of io_uring into MPICH

We add an io_uring-based backend to ROMIO's existing non-blocking MPI-IO path; the MPI-IO API and semantics remain unchanged



Integration of io_uring into MPICH

- ROMIO prepares an SQE (`prep_write`), enqueues it in the shared submission queue, and submits the batch via `io_uring_enter`, kernel I/O workers execute the I/O.
- The app receives an `MPI_Request` immediately, compute while kernel workers handle I/O; no userspace helper threads.
- On `MPI_WaitAll`, ROMIO drains the completion queue entries via `wait_cqe/peek_cqe`, tagged CQEs give O(1) work per completion, O(N) total, versus POSIX AIO's $O(N^2)$ aiocb scanning



Contribution

io_uring backend keeps the MPI-IO API intact, but replaces POSIX AIO's helper-thread + $O(N^2)$ completion path with batched SQEs and O(N) tagged completions, aligning ROMIO with NVMe's parallel interface.

Evaluation

Platform

- Single-node
- i7-12700 (12 cores / 20 threads)
- 128 GiB DDR5
- Samsung 980 PRO 1 TB NVMe (PCIe 4.0)
- MPICH 4.3.0
- liburing 2.x

Methodology

Backends compared:

- Blocking (`pread/pwrite`)
- POSIX AIO non-blocking (`MPI_File_iread_at`)
- io_uring-based non-blocking backend

- O_DIRECT, page cache dropped between runs to expose device behaviour
- POSIX AIO would otherwise degenerate to synchronous behavior on cache misses

Workloads

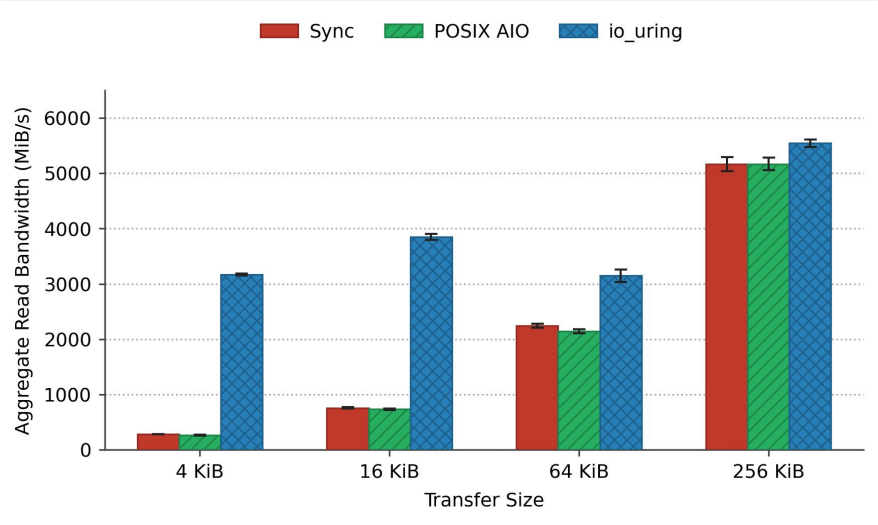
- **Synthetic**: IOR – configurable transfer size, request count, and access mode, used to isolate submission overhead and concurrency effects.

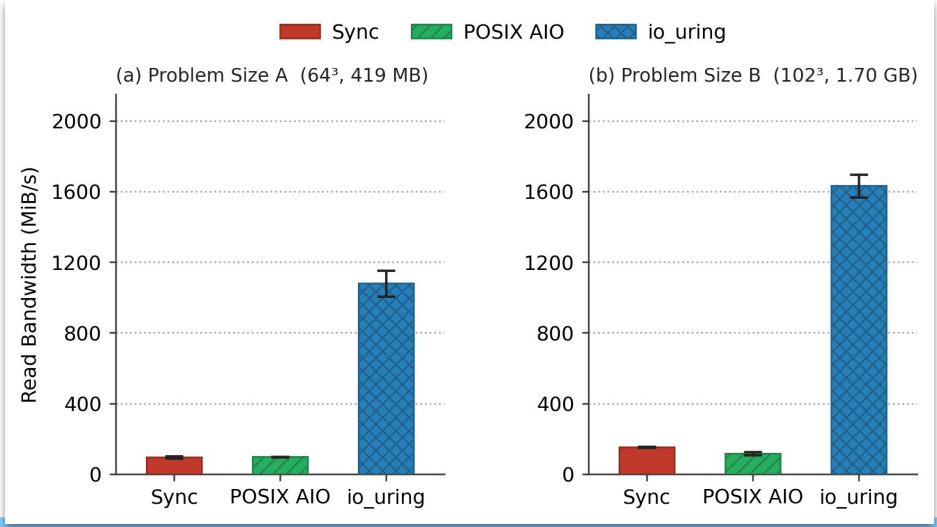
- **Burst-heavy apps**:
  - BT-IO (NAS BT CFD I/O phase) → bursts of independent writes/reads per rank, thousands of concurrent operations.
  - WaComM++ (marine pollutant transport) → asynchronous MPI-IO used to study compute–I/O overlap.

- **Low-concurrency checkpointing**: HACC-IO – nine independent writes per checkpoint and corresponding verification reads, stressing low in-flight concurrency

Where io_uring helps: bursty, small reads on local NVMe

Workload	Condition	Speedup
IOR	4KiB, 128 ops / rank, 4 ranks.	11.2x vs sync 12.0x vs AIO
BT-IO	Class A: 2048 concurrent read ops Class B: 5202 ops. Upto 208,080 ops for certain phases.	Class A: 11.2x vs sync, 11.0x vs AIO Class B: 10.7x vs sync, 14.1x vs AIO





Where io_uring helps: Compute–I/O overlap

- WaComM++ with asynchronous MPI-IO and variable compute intensity:
  - For reads, io_uring's wait time drops dramatically as compute intensity increases (up to ~94% reduction vs AIO), showing effective overlap: most I/O completes while the app computes.
  - POSIX AIO's read wait times remain nearly constant regardless of compute, because helper threads contend with compute threads for CPU and progress less during the compute phase.
- For writes, both backends see near-zero wait time once data hits the NVMe controller's write buffer (O_DIRECT does not bypass that), so overlap does not change write completion latency

Size	Compute Intensity	Phase	Δ (%)
1024	0	Read	-50.2%
		Write	+0.7%
	30	Read	-94.1%
		Write	0.0%
	60	Read	-94.3%
		Write	-5.2%
	90	Read	-94.1%
		Write	0.0%
16384	0	Read	-34.8%
		Write	-4.7%
	30	Read	-94.1%
		Write	+10.3%
	60	Read	-94.4%
		Write	-11.6%
	90	Read	-94.3%
		Write	-0.6%
131072	0	Read	-21.9%
		Write	-11.9%
	30	Read	-92.5%
		Write	+6.3%
	60	Read	-91.8%
		Write	+2.4%
	90	Read	-91.0%
		Write	+2.5%

Where async I/O does not provide benefits

- **Parallel filesystem (BeeGFS)**: repeating the IOR sweep on BeeGFS, all three backends deliver indistinguishable read bandwidth, network round-trip latency dominates, leaving no room for submission-path optimization.

- **Low-concurrency app (HACC-IO)**: only nine concurrent requests per checkpoint, worst-case AIO completion scanning is at most 81 `aio_error` calls, so io_uring's more efficient completion path brings no measurable throughput difference.

- **Writes on NVMe**: across all experiments, write bandwidth is indistinguishable across backends; O_DIRECT bypasses the page cache but not the NVMe controller's internal write buffer, so writes are acknowledged as soon as they reach the controller, regardless of submission efficiency.

Conclusion

These results pinpoint the regime where kernel I/O interface choice matters: read-heavy, bursty, high-concurrency MPI-IO to local NVMe, not large sequential transfers, writes, low concurrency, or networked storage

Future work

- **Transparent adoption of non-blocking MPI-IO**
  - Today, HPC applications must be manually modified to use non-blocking MPI-IO.
  - A natural next step would be to build a transparent interception layer that replaces blocking and existing non-blocking MPI-IO calls with io_uring-backed implementations, without changing application code.

- **Workload-aware batching into SQ rings**
  - Combine this with workload characterization (e.g., Darshan traces or lightweight online detection) to detect bursty patterns and batch individual calls into SQ ring submissions
  - Such a framework would make io_uring's advantages accessible to the large body of HPC applications that still rely on blocking POSIX I/O.

THANK YOU!

Questions?