PDSW'21

Verifying IO Synchronization from MPI Traces

S. Yellapragada, C. Wang, M. Snir

University of Illinois Urbana-Champaign

Motivation

- Most HPC applications use POSIX compliant Parallel File Systems
- POSIX Semantics: *write* will be immediately visible to *read*
  -- Provided accesses are not concurrent
- Concurrent events could return partial results
  -- Likely bug in your application or I/O library
- Answer this question - "Are I/O operations of HPC applications properly synchronized?"

Approach

1. **Idea:** Leverage the order guaranteed by MPI Standard to generate "Happens-Before" between I/O operations and verify the synchronization.
2. MPI Communications:
   - MPI_SEND() ---> MPI_RECV()
   - MPI_BCAST()
3. "Happens-Before" relation is the transitive closure of the order imposed by the semantics of MPI communication operations and the program order within each process.
4. Two I/O operations are synchronized if they are ordered by this happen-before order.
5. Tracing Tool: **Recorder** - github.com/uiuc-hpc/Recorder
6. Traces from 17 Scientific applications – https://library.ucsd.edu/dc/object/bb95276921

Design

Generate Trace Records

- Generate **Trace** files
- **Identify the required calls - (Rank No., Index No.)**

Generate Causality Order

- Matching Algorithm
- DAG to represent Happens-Before Order

Verification Process

- Detect Conflict Accesses
- Check for synchronization

Phase 1

- Recorder generates one file each for the process
- Includes MPI-IO, HDF5, POSIX
- The records contain the values of all of the parameters supplied to those operations, e.g., for I/O, file name, offset, and flags.

T_start	T_end	MPI_Call	*buffer*	*count*	MPI_Datatype	*source/ dest*	*tag*	Communicator	*request*	*status*
315	343	MPI_Isend	0x7fffffff63b0	4	MPI_BYTE	0	123	MPI_COMM_WORLD	0x7fff0af44dd8	--
346474	346525	MPI_Recv	0x7fff0af44e00	10	MPI_INT	1	1	MPI_COMM_WORLD	--	0x7fff0af44dd0
353	2559	MPI_Wait	--	--	--	--	--	--	0x7fff0af44dd0	0x7fff0af44de0
13755	14654	MPI_Bcast	0x7ffda5ebc034	1	MPI_INT	0	--	MPI_COMM_WORLD	--	--

- Each record can be uniquely identified by the file containing the record, and by the sequence number of the record - (*Rank No, Sequence id*)

Phase 2

Point-to-Point

```
MPI_Send(const void *buf, int count,
MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)

MPI_Recv(void *buf, int count,
MPI_Datatype datatype, int source, int
tag,  MPI_Comm comm, MPI_Status *status)
```

If call == MPI_Send:
- Read Destination
- Go to Destination Rank:
  - Find MPI_Recv with source = dest

Collective

```
MPI_Bcast( void *buffer, int count,
MPI_Datatype datatype, int root,
MPI_Comm comm )
```

If call == MPI_Bcast:
- Read Root
- For every rank in the communicator:
  - Find Bcast with root = Root

Non-blocking

```
MPI_Irecv(void *buf, int count,
MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Request *request)

MPI_Wait(MPI_Request *request, MPI_Status
*status)
```

For a matching Irecv:
- Read Request
- Continue surfing calls in this rank
  - Wait/Waitall with request = Request

Implementation

recv_call_ids

1	2

Coll_call_ids
Key:(func_name, comm)
Value: list of ids

(MPI_Barrier, comm1)	[4,7]
(MPI_Ialltoall, comm2)	[5]

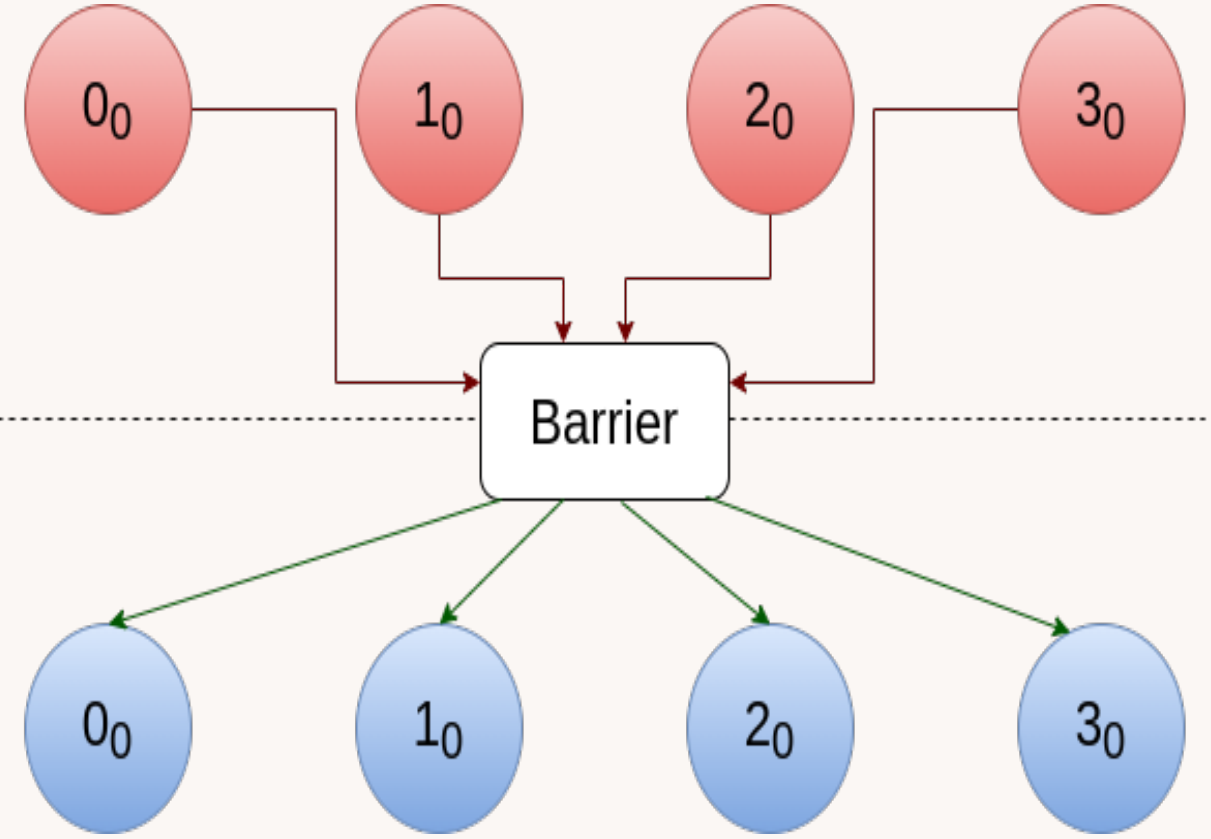
wait_test_call_ids

6	8

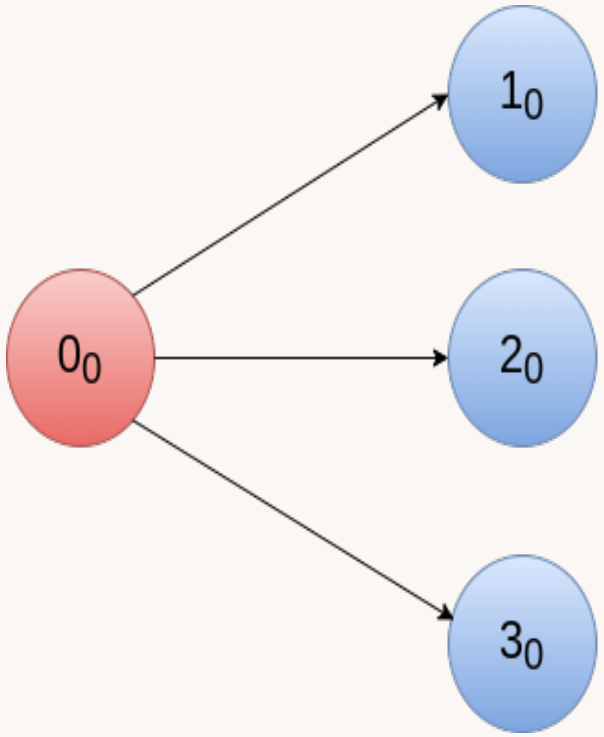
all_calls

0	MPI_Send(dts, tag, comm1)
1	MPI_Recv(src, tag, comm1)
2	MPI_Recv(src, tag, comm1)
3	MPI_Isend(dst, tag, comm2, req1)
4	MPI_Barrier(comm1)
5	MPI_Ialltoall(comm2, req2)
6	MPI_Wait(req1)
7	MPI_Barrier(comm1)
8	MPI_Test(req2)

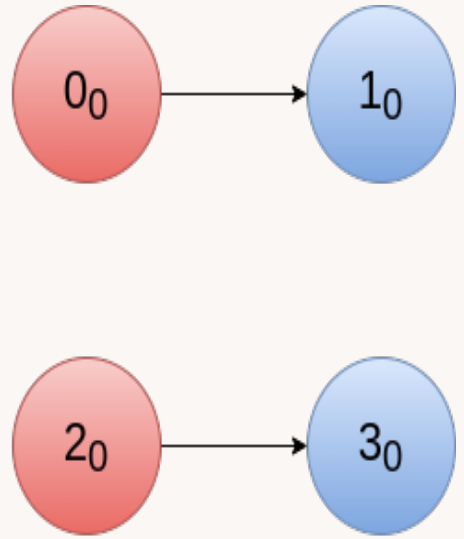
- **All_calls:**
  - Calls from Phase 1
- **Coll_call_ids:**
  - Hashtable:
    - Key: MPI_COLL call
    - Value: Index in the list
- **Recv_call_ids:**
  - Index of Recv calls in the list
  - Prevents reiterating over already matched calls
- **Wait_test_call_ids:**
  - Similar to Recv_call_ids



a) MPI_Barrier



b) MPI_Bcast



c) MPI_Send/Recv

Phase 3

- Algorithm from prior work[1] to detect **conflicting I/O** operations:
  - RW-[S|D]: read conflicting with a write by the same process (S) or by different (D) processes.
  - WW-[S|D]: write conflicting with another write by the same process (S) or by different (D) processes.
- For each of the detected conflict pair – verify from the Happens before order Verify the order of conflicting pairs from the DAG.

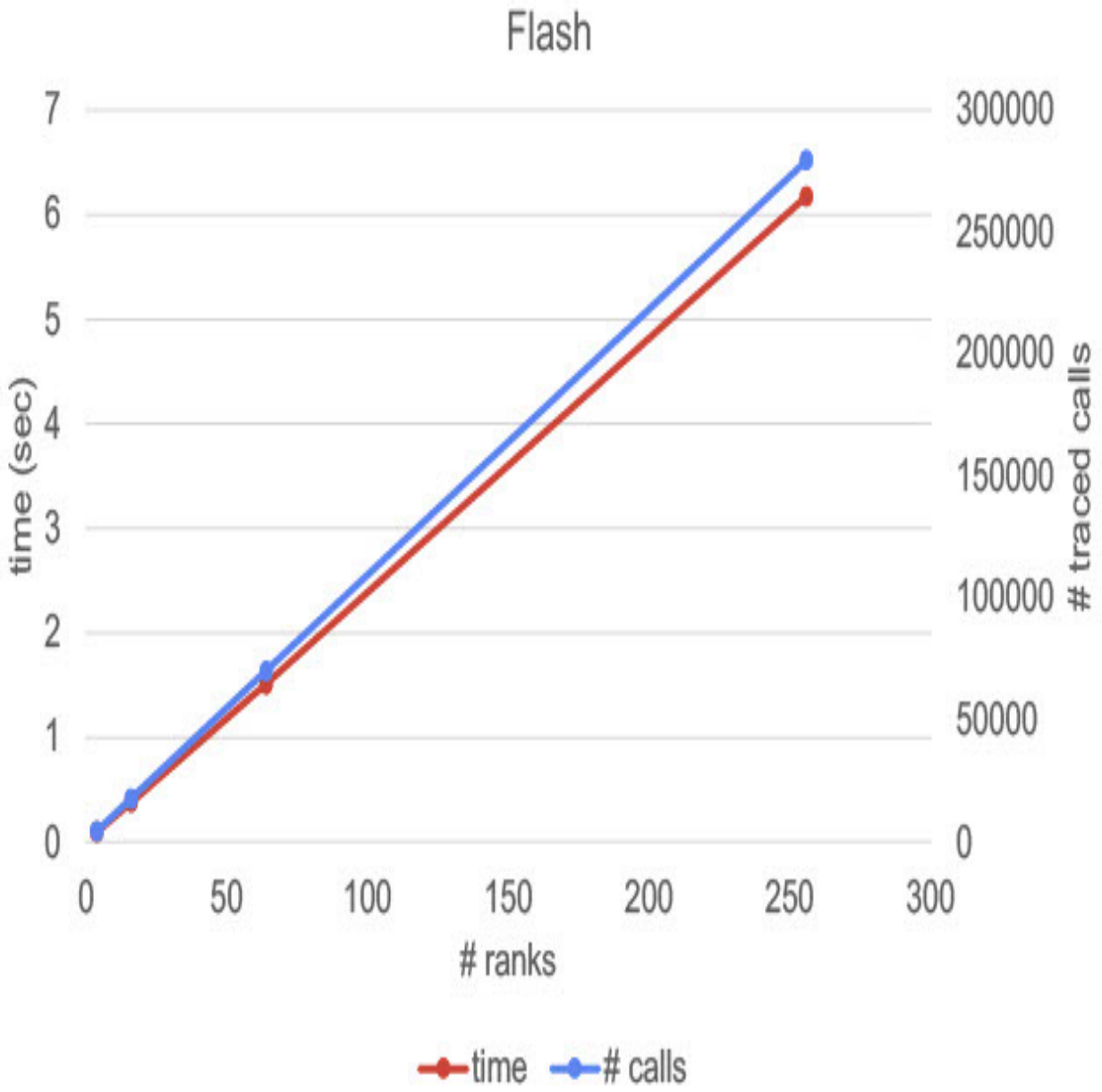
[1] C. Wang, K. Mohror, and M. Snir, "File system semantics requirements of HPC applications," in Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, HPDC'21, NY, USA: Available: https://doi.org/10.1145/3431379.3460637

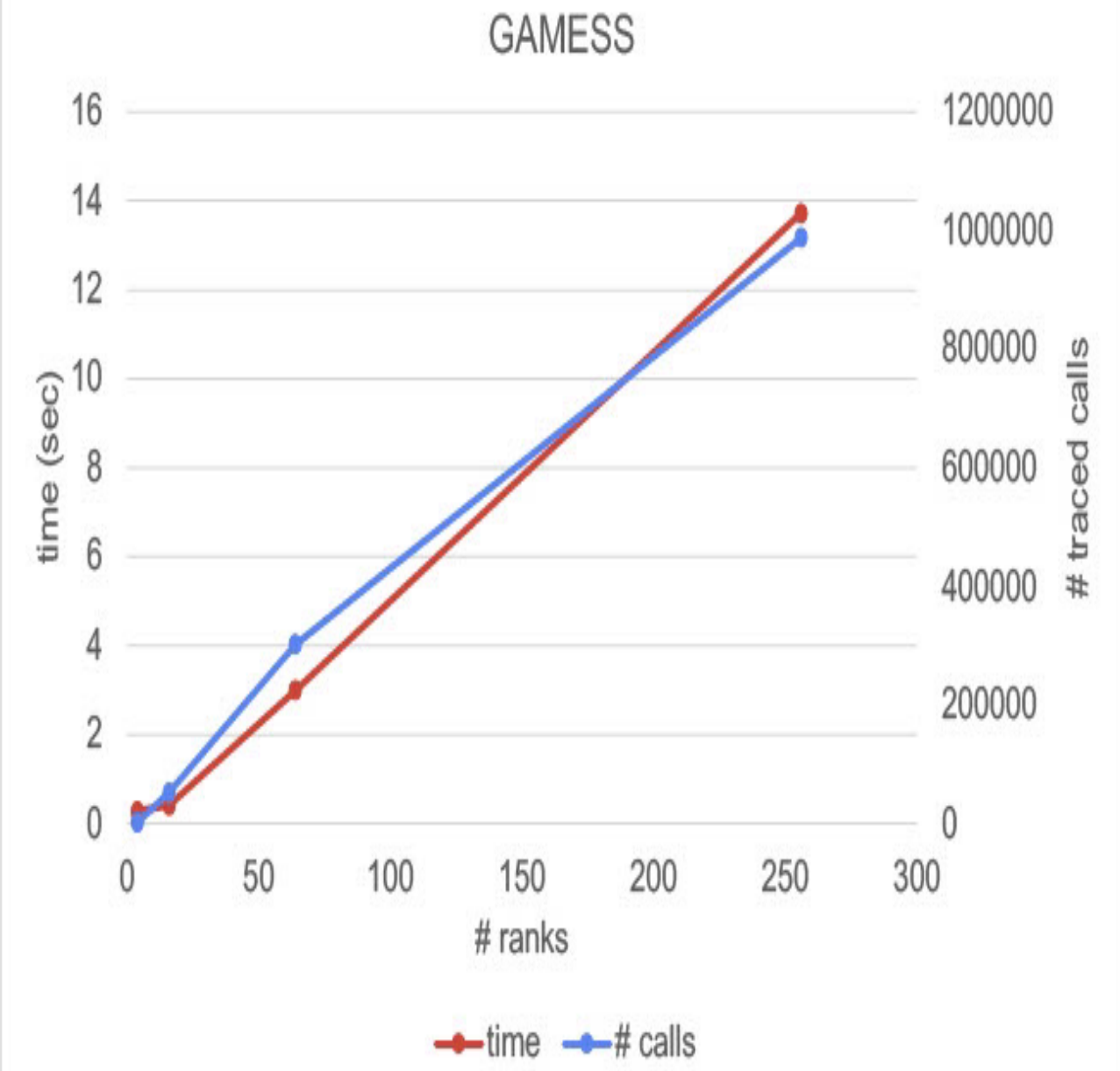
Experiments and Results

All the experiments to study tool's scalability and perfromance were performed on:

- An Intel x86_64 architecture machine with
- 2-core/4-thread 2.90 GHz Intel i5-5300U processor
- 8GB main memory running Ubuntu 21.04 operating system and
- Python version 2.7.18.

EVALUATION


Flash
time (sec)
traced calls
ranks
0
1
2
3
4
5
6
7
0
50000
100000
150000
200000
250000
300000
0
50
100
150
200
250
300
time
calls


GAMESS
time (sec)
traced calls
ranks
0
2
4
6
8
10
12
14
16
0
200000
400000
600000
800000
1000000
1200000
0
50
100
150
200
250
300
time
calls

Observations

- The number of recorded calls increases linearly with the number of ranks (weak scaling)
- Our algorithm's execution time increases linearly as the numberof MPI ranks
- The number of nodes and edges is roughly proportional to the number of calls
- Querying the DAG to verify synchronization - a millisecond per conflicting pair.

Application	I/O Library	WW		RW		Properly Synchronized
		S	D	S	D	
FLASH	HDF5	✓	✓			✓
ENZO	HDF5			✓		✓
NWChem	POSIX	✓		✓		✓
pF3D-IO	POSIX			✓		✓
MACSio	Silo	✓				✓
GAMESS	POSIX	✓				✓
LAMMPS	ADIOS	✓				✓
LAMMPS	NetCDF	✓				✓

Conclusion

- In this work, we presented a tool to verify that IO operations in HPC codes are properly synchronized.
- Total of 17 applications were studied:
  - 10 applications – No conflicts
  - 7 applications – Conflicting I/O accesses were properly synchronized

Applications:

- Programmers can use the tool to check that their IO code is race-free.
- The IO of HPC applications to be race-free and to use this assumption in the design of Parallel File Systems.
- The timestamp order of conflicting file accesses matched the happens-before order

PDSW'21

THANK YOU!

S. Yellapragada, C. Wang, M. Snir

University of Illinois Urbana-Champaign