2021 IEEE/ACM Sixth International Parallel Data Systems Workshop (PDSW) | 978-1-6654-1837-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/PDSW54622.2021.00012

Verifying IO Synchronization from MPI Traces

Sushma Yellapragada, Chen Wang, Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois, 61801-2302
{sushmay2, chenw5, snir}@illinois.edu

***Abstract*—The paper addresses the following question: Are IO operations of HPC applications properly synchronized? We focus on parallel file systems that satisfy POSIX semantics. The outcome of I/O operations is well-defined provided that conflicting accesses to a file location are not concurrent, but are ordered. Accesses to distinct processes are ordered by the executed MPI communication. We derive the "happens-before" relation between I/O calls of HPC runs by analyzing traces collected during program execution. Various optimizations reduce the analysis overhead. We collected traces from 17 representative HPC applications. We found that 10 of them do not perform conflicting I/O accesses and, hence, are properly synchronized by default. The remaining 7 applications properly synchronize the conflicting I/O accesses.**

I. Introduction

Most HPC applications use POSIX compliant Parallel File Systems (PFS), such as Lustre [1] or GPFS [2]. These are either accessed directly by the application, using POSIX I/O calls, or are accessed indirectly, through parallel I/O libraries. POSIX semantics require that the value returned by a read to a file location is the value written to that location by the last preceding write [3]. However, reads and writes are not required to be atomic: a read that is concurrent with a write executed by another process could return part of the written values [4]. (On the other hand, reads and writes executed by distinct threads of the same process are not interleaved [5].)

The lack of atomicity is a reasonable choice: It avoids the need for locks that could hamper performance for long reads or writes; and an application pattern where the programmer does not care in what order two conflicting file accesses execute is unlikely in HPC applications. Lack of proper synchronization is most likely a bug in the application or I/O library. Thus, we expect that HPC I/O is *race-free* – i.e., that conflicting I/O operations are not concurrent.

We hypothesize that the I/O code of HPC applications is race-free. The main motivation for our work is to verify this hypothesis. This would enable various optimizations in the design of PFS's – similar to the advantages of data race-free codes for which sequential consistency can be efficiently enforced on weakly consistent hardware [6]. In our previous work on the Recorder tool [7] we assumed that the global wall-clock order of conflicting file accesses matched the happens-before order; the former is much easier to capture than the later. The assumption holds for race-free code. The results of the current paper validate our assumption.

In a distributed system the *happens-before* relation is a partial order that is determined by inter-process synchronizing communications. In HPC, these communications use MPI. For example, if a call to `MPI_Send()` on one process is matched by a call to `MPI_Receive()` on another process, then the start of the send call happens before the completion of the receive call; in a collective call to `MPI_Reduce()`, the start of the call at any participating process happens before the completion of the call at the root. If the processes of an MPI program communicate only using MPI, then happens-before (aka causality) order of the execution is the transitive closure of the order imposed by the semantics of MPI communication operations and the program order within each process; two I/O operations are synchronized if they are ordered by this happen-before order.

We designed a tool to check whether HPC applications are "I/O race-free" and used it to study representative HPC applications. We describe in the following section the overall design of our tool. This is followed by a section discussing the implementation of the tool. in Section IV by a description of our findings for 17 representative HPC applications and a discussion of the tool's performance. Related work is discussed in Section V, followed by a conclusion in Section VI.

II. Design

The analysis of I/O synchronization consists of the following phases:

1) Trace records are created for all I/O and MPI calls.
2) An offline analysis of the trace records generates a graph representing all executed MPI and I/O calls and the happens-before relation between them. Another offline analysis identifies conflicting I/O operations.
3) The results of the two analyses are combined to check whether conflicting I/O operations are properly ordered.

A. Tracing

We use a trace-driven analysis approach, rather than on-the-fly causality analysis since the same traces are also used to identify conflicting I/O calls, and for other types of analyses. We use the Recorder tracing tool [7] to collect traces from applications. Recorder stores a record for every MPI call and for every I/O operation during an application run, including I/O operations from MPI-IO, HDF5 and 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. Trace

978-1-6654-1837-9/21/$31.00 ©2021 IEEE
DOI 10.1109/PDSW54622.2021.00012

files are numbered according to the rank of the corresponding process in `MPI_COMM_WORLD`. Each record can be uniquely identified by the file containing the record, and by the *sequence number* of the record.

B. Trace Analysis

1) The Happens-Before Order: The semantics of MPI imposes some order between related communication operations. Their main cases are listed below.

a) Blocking point-to-point communications: This simple case is illustrated in Figure 1a: If a send matches a receive, then the send starts before the receive returns. To identify this order, we need to properly match send and receive records.

b) Nonblocking communications: The case of a nonblocking communication, illustrated in Figure 1b is slightly more complex: If a nonblocking send matched a nonblocking receive, and the receive is completed by a wait (or test) call, then the send starts before the wait returns. To identify this order, we need to properly match isend to irecv and irecv to wait.

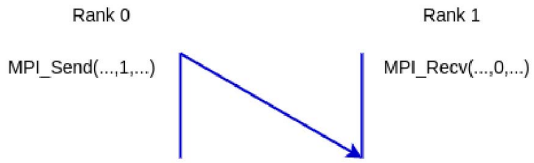
c) Rooted collective communications: Figure 1c illustrates a broadcast operation, where the root is the sender. No caller in the involved group can return from the collective call before the root entered the call. Figure 1d illustrates the reverse case of a reduce operation, where the root is the receiver. The root cannot return from the call before the every process in the involved group entered the call.

d) Unrooted collective communication: The final case, illustrated in Figure 1e, is that of a symmetric collective communication, such as barrier: No process in the involved group can exit the call before every process in the group has entered it.

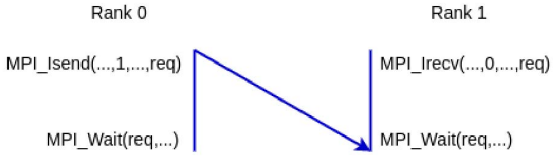
2) Computing the Happens-Before Relation: The happens-before order among MPI and I/O calls is described by a Directed Acyclic Graph (DAG): The nodes correspond to trace records and the edges corresponds to happens-before relations. An edge from record a to record b indicates that a started before b completed. We assume that each MPI process is single-threaded, so that records in the same trace file are totally ordered. In addition, MPI communications order records from different files. The complete happens-before relation will be obtained by computing the transitive closure of this DAG.

Typically, the number of pairs of conflicting I/O operations is much smaller than the number of MPI operations. Also, typically, the number of edges in the DAG is linear in the number of records, while the number of edges in the transitive closure graph is quadratic in the number of records. Therefore, rather than computing the transitive closure of the DAG, we shall query, for each conflicting pair of I/O operations, whether the DAG contains a directed path leading from one to the other.

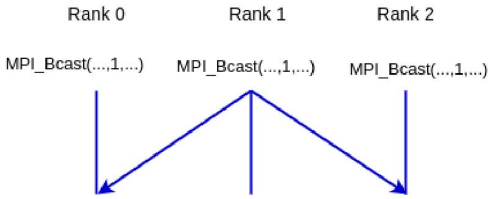
We shall focus now on the construction of the happens-before DAG. The algorithm handles the four "happens-before" patterns discussed in the previous section; it also takes into account the semantics of MPI and, in particular, the ordering constraints: point-to-point messages are matched in order and collective operations are invoked in order.



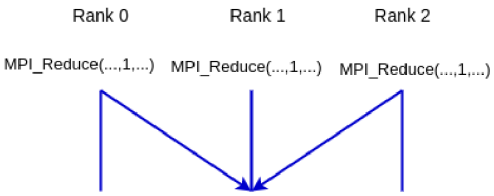
(a) Blocking point-to-point



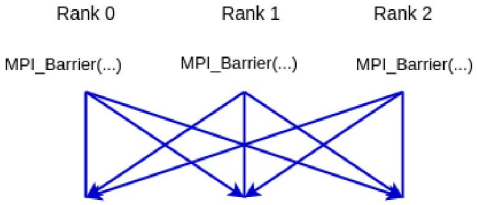
(b) Nonblocking point-to-point



(c) Broadcast



(d) Reduce



(e) Barrier

Fig. 1: Order imposed by MPI

Consider first the simple case of blocking sends and receives. We match sends to receives by traversing each file of records in order: The first unmatched record of a send in the file of process A with a destination of B will be paired with the first unmatched record of a receive in the file of process B with a source of A and the same tag. An edge connecting the send record to the receiver record will be inserted. Because of the MPI ordering constraints, this pairing process will match correctly sends to receives.

Receives that use `MPI_ANY_SOURCE` do not provide an input source argument that can be used to match with the appropriate sender. Consider the communication pattern illus-

Fig. 2: Nondeterministic code using wildcard receives

trated in Figure 2: Without the source value, it is not possible to know which receive matched which send, as this depends on the order of arrival of the sent messages. However, the sender rank can be retrieved from the status output argument of the receive call. The Recorder tracing tool captures the value of this argument. Thus, for non deterministic MPI code, our tool recreates the particular ordering in which communications occurred.

In order to handle nonblocking receives we need to match the record of the `MPI_Irecv` call with the subsequent record of the `MPI_Wait` or `MPI_Test` call that completed the receive operation. This will be the first wait or the first successful test that uses the same request as the receive call. An edge will be inserted that connects the record of the send call to the record of that wait or test call. One slight complication is that, if the nonblocking receive call uses a wildcard source argument, then the actual source value will be found in the record of the matching wait or test operation.

The handling of collective operations is similar: The first unmatched record for a collective call will be paired with the first unmatched record for the same call at all the other ranks involved in the call. For rooted collectives, an edge will connect the record of the root call to all the other records, or vice versa. For unrooted blocking collective calls, we need to logically separate the start of the call and the end of the call, and insert an edge from each start to each end. Rather than creating n^2 edges, we create $2n$ edges by inserting a node that represents the collective call.

MPI communication operations have a communicator argument and MPI ranks are relative to the group of this communicator. In order to properly pair sends to receives, or all matching invocations of a collective function, we need to keep track of the communicator groups and the match between absolute ranks (i.e., the rank in `MPI_COMM_WORLD`) and relative ranks for the communicators created during program execution. The can be done during the offline processing of the records for communicator creating calls, since these records store all the values of all the arguments passed to the communicator constructor.

C. Examine conflicts

We used an algorithm from our prior work [8] to detect conflicting I/O operations and then check for synchronizations. The post-mortem analysis tools offered by Recorder can answer questions such as whether an application performs conflicting updates to the same file location. We define the following cases as *potential* conflicts:

- 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.

III. Implementation

In this section, we provide more details on the Python code used to implement the tool. Once the application traces are generated using Recorder, our program reads each trace file to extract various I/O operations executed by the application. Each extracted record contains *(rank, index, call, src, dst, stag, rtag, comm, tindx, req, reqflag)* where *rank* is used to identify the MPI process, *index* is the order in which the call appeared in the application, *src* and *stag* are the rank and the tag of the process who made the call respectively, while *dst* and *rtag* are the rank and the tag of the receiving process respectively, *call* indicates the MPI routine, and *comm* is the MPI communicator the call belongs to. *req, tindx* are parameters related to non-blocking calls referring to the *request*, and index of the completed requests in *array_of_requests* for `MPI_Waitall`, `MPI_Waitsome`, etc.

In addition, three auxiliary arrays are used specifically for the following three types of MPI calls: collective calls, point-to-point receive calls, and wait or test calls. They are used to accelerate searching during the matching process.

- **recv_calls**: Contains indices of point-to-point receive calls like `MPI_Irecv`, `MPI_Recv`, etc. When matching a point-to-point send call, one iterates this array instead of all trace records. Upon a successful match, the call will be deleted from this array.
- **coll_calls**: Contains all collective calls, including `MPI_Barrier`, `MPI_Alltoall`, etc. Collective calls are matched in order and only two fields are required to match: communicator and function name. Thus, we use a hash table to store them wherein each entry, the key is its communicator and the value is its index. As a result, a collective call can be matched in O(1) time for each rank.
- **wait_test_calls**: This array contains indices of all `MPI_Waitxxx/MPI_Testxxx` calls. When a non-blocking receive call is being matched, an edge is created between the send call and the wait/test call that completes the receive's request, as shown in Figure 1(b). Consequently, this request will be removed from the completed request list of the corresponding wait/test call.

We traverse each array to find a match for each type of MPI routine as described in Section II-B. Code for matching point-to-point calls and collective calls is shown in Listing 1. `get_global_rank()` translates a rank in any communicator to the rank in `MPI_COMM_WORLD`. `find_wait_test_call()` finds fist wait/test call that completes the request from a non-blocking call, with the help of wait_test_calls array.

Each matching pair resulting from this algorithm is essentially the head and tail of an edge that represents the "happens-before" order. For example, in a Send-Recv pair,

Listing 1 Algorithm to find matching pairs

```
def match_pt2pt(send_call)
    head, tail = [send_call], []
    dst = get_global_rank(send_call.comm, \
                    send_call.dst, comm_table);

    for each idx in recv_calls[dst]:
        recv_call = all_calls[dst][idx];
        if (not same tag) or (not same comm) \
                          or (not same src):
            continue;

        recv_calls[dst].remove(idx);
        if recv_call.is_blocking_call():
            add_edge(head, tail, recv_call);
            break
        else:
            wt = find_wait_test_call(req);
            add_edge(head, tail, wt);
            break

def match_coll(coll_call):
    head, tail = [], []
    key = coll_call.get_key();
    for each rank:
        idx = coll_calls[key][0];
        mycall = all_calls[idx];
        coll_calls[key].remove(idx);

        if (mycall.is_blocking_call()):
            add_edge(head, tail, mycall);

            if(coll_calls[key].empty())
                coll_calls.remove(key);
        else:
            wt = find_wait_test_call(mycall.req);
            add_edge(head, tail, wt);
```

an edge between two nodes `MPI_Send` → `MPI_Recv` is added to the graph. Once the graph is generated, we use the procedure described in Section II-C to identify potentially conflicting pairs of of POSIX I/O calls. Nodes for these calls are subsequently added to the "happens-before" DAG. Finally, for each such pair, if there exists a path from one node to the other, then they are properly synchronized. We use the *networkx* network analysis library [9] to perform this test.

IV. Results

We use the same set of traces collected from our previous work [8]. Those traces were collected from 17 HPC applications and I/O benchmarks that span a wide variety of domains. Among the 17 applications, 7 exhibited potential conflicting I/O accesses. We run our analysis tool on the traces from these 7 applications. All the experiments were performed on an Intel x86_64 architecture machine with 4 Intel(R) Core(TM) i5-5300U processors and 8GB main memory running Ubuntu 21.04 operating system and Python version 2.7.18.

A. Are conflicting accesses properly synchronized?

From the results reported in Table I it can be confirmed that our assumption: all conflicting accesses are indeed properly synchronized is true. Note that 7 applications performed conflicting I/O accesses, and only one (FLASH) among them involved conflicting accesses from distinct processes.

TABLE I: Potential conflicting accesses and whether they are properly synchronized. 'S' indicates conflicting operations called by the same process; 'D' indicates that the conflict involves multiple processes.

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	✓				✓
LAMMPS	HDF5					
LAMMPS	MPI-IO					
LAMMPS	POSIX					
MILC-QCD	POSIX					
ParaDiS	HDF5					
ParaDiS	POSIX					
VASP	POSIX					
LBANN	POSIX					
QMCPACK	HDF5					
Nek5000	POSIX					
GTC	POSIX					
Chombo	HDF5					
HACC-IO	MPI-IO					
HACC-IO	POSIX					
VPIC-IO	HDF5					

B. Performance

This section discusses how our tool performed against different applications with varying trace sizes. Table II lists the number of recorded calls and the numbers of nodes and edges in the computed DAG for three applications, with a varying number of MPI ranks. The number of recorded calls increases linearly with the number of ranks (weak scaling) and the number of nodes and edges is roughly proportional to the number of calls: Our DAG representation is efficient.

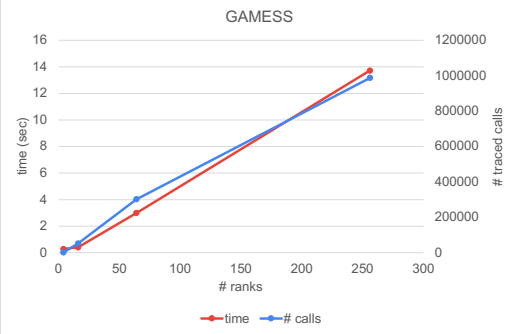
We show in Figure 3 the time required to compute the happens-before DAG for three applications, as a function of the number of MPI ranks (red line) and the total number of MPI calls captured by Recorder for each run (blue line). Our algorithm's execution time increases linearly as the number of MPI ranks, which is expected as the matching time is proportional to the number of MPI calls. Lastly, querying the DAG to verify synchronization only takes on average a millisecond per conflicting pair.

V. Related Work

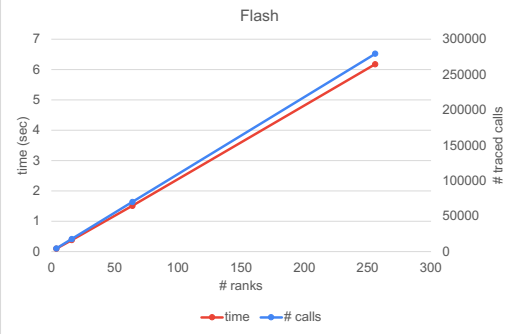
Given the fundamental importance of parallel computing to science and engineering research, application performance and correctness is paramount. MPI is widely used in HPC applications scaling to thousands of cores. Some of the previous works like Paraver [10] offer fine-grained performance metrics and visualizations, but their accurate interpretation requires a substantial time and effort from highly-skilled analysts. The concept of the detection of causal execution flows for cause-effect inference has been recently studied in an automated

TABLE II: Size of of the generated "happens-before" DAG

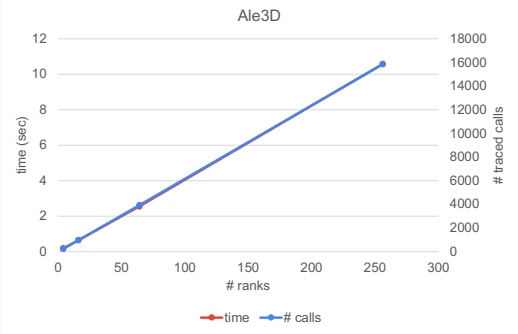
Ranks	GAMESS			FLASH			MACSio		
	Calls	Nodes	Edges	Calls	Nodes	Edges	Calls	Nodes	Edges
4	1657	1596	2530	4626	2760	4048	232	140	172
16	52590	7146	13900	17770	11188	17586	960	592	772
64	301776	201420	317346	70138	44692	71394	3936	2464	3268
256	987784	982700	1537548	279610	178708	286626	15840	9952	13252



(a) GAMESS - a quantum chemistry application



(b) FLASH - a multi-physics simulation code



(c) MACSio - an Ale3D proxy application

Fig. 3: Performance evaluation for 3 HPC applications

problem diagnosis thesis [11]. Online causality analysis [12] focus on runtime discovery of causal execution flows, made up of communication and computational activities in message-passing parallel programs. For example, Large Scale Verification of MPI Programs [13] talks about dynamically tracking causality orders with a lazy update for non-blocking calls but involves overheads in terms of communication bandwidth, latency, and memory consumption. Such online analysis and causality tracking protocols that rely solely on logical clocks fail to capture all nuances of MPI program behavior, including the non deterministic nature in which non-blocking calls can complete. Another work on distributed wait state tracking [14] is a run time error detection tool that provides a wide variety of automatic correctness checks. It was designed especially for deadlock detection and works by simulating MPI blocking semantics.

Several automatic performance analysis tools have been developed, for example KappaPI [15] and EXPERT [16] that perform offline analysis of event traces searching for patterns that indicate an inefficient behavior. These tools take a trace file from the execution of the application and try to detect performance bottlenecks using certain performance properties. However a limitation with these tools is they do not provide any insight into internal consistency protocols that could effect application correctness. Existing (shared-memory concurrent program) debugging techniques [17] do not directly carry over to MPI, where operations typically match and complete out-of-program order according to an MPI-specific matches-before order.

VI. Conclusion

In this work, we presented a tool to verify that IO operations in HPC codes are properly synchronized and used this tool to analyze representative HPC applications. Application programmers can use the tool to check that their IO code is race-free. The analysis indicates that it is reasonable to require the IO of HPC applications to be race-free and to use this assumption in the design of Parallel File Systems.

Acknowledgment

This work was supported by NSF grant CCF 17-63540.

References

[1] S. Cochrane, K. Kutzer, and L. McIntosh, "Solving the "hpc i/o" bottleneck: "sun lustre" storage system," *Sun BluePrints Online, Sun Microsystems*, 2009.

[2] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters." in *FAST*, vol. 2, no. 19, 2002.

[3] IEEE, "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7," *1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, 2018.

[4] G. Ntzik, P. da Rocha Pinto, J. Sutherland, and P. Gardner, "A concurrent specification of POSIX file systems," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[5] IEEE, "Threads Extension for Portable Operating Systems (Draft 6)." *IEEE Std. P1003.4a/D6.*, February, 1992.

[6] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill, "Programming for different memory consistency models," *Journal of parallel and distributed computing*, vol. 15, no. 4, pp. 399–407, 1992.
[7] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel I/O tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 1–8.
[8] 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*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2020, p. 19–30. [Online]. Available: https://doi.org/10.1145/3431379.3460637
[9] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using NetworkX," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.
[10] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1. Citeseer, 1995, pp. 17–31.
[11] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 88–es. [Online]. Available: https://doi.org/10.1145/1188455.1188548
[12] O. Morajko, A. Morajko, T. Margalef, and E. Luque, "On-line performance modeling for MPI applications," in *Euro-Par 2008 – Parallel Processing*, E. Luque, T. Margalef, and D. Benítez, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 68–77.
[13] A. Vo, G. Gopalakrishnan, R. M. Kirby, B. R. de Supinski, M. Schulz, and G. Bronevetsky, "Large scale verification of mpi programs using lamport clocks with lazy update," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 330–339.
[14] T. Hilbrich, B. R. de Supinski, W. E. Nagel, J. Protze, C. Baier, and M. S. Müller, "Distributed wait state tracking for runtime MPI deadlock detection," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.
[15] J. Jorba, T. Margalef, and E. Luque, "Performance analysis of parallel applications with KappaPI 2." in *PARCO*. Citeseer, 2005, pp. 155–162.
[16] F. Wolf and B. Mohr, "Automatic performance analysis of hybrid MPI/OpenMP applications," *Journal of Systems Architecture*, vol. 49, no. 10-11, pp. 421–439, 2003.
[17] M. Russinovich and B. Cogswell, "Replay for concurrent non-deterministic shared-memory applications," in *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, 1996, pp. 258–266.