An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs

Simon Schwitanski
IT Center
RWTH Aachen University
Aachen, Germany
schwitanski@itc.rwth-aachen.de

Felix Tomski
IT Center
RWTH Aachen University
Aachen, Germany
tomski@itc.rwth-aachen.de

Joachim Protze
IT Center
RWTH Aachen University
Aachen, Germany
protze@itc.rwth-aachen.de

Christian Terboven
IT Center
RWTH Aachen University
Aachen, Germany
terboven@itc.rwth-aachen.de

Matthias S. Müller
IT Center
RWTH Aachen University
Aachen, Germany
mueller@itc.rwth-aachen.de

***Abstract*—Vector clocks are logical timestamps used in correctness tools to analyze the happened-before relation between events in parallel program executions. In particular, race detectors use them to find concurrent conflicting memory accesses, and replay tools use them to reproduce or find alternative execution paths. To record the happened-before relation with vector clocks, tool developers have to consider the different synchronization concepts of a programming model, e.g., barriers, locks, or message exchanges. Especially in distributed-memory programs, various concepts result in explicit and implicit synchronization between processes. Previously implemented vector clock exchanges are often specific to a single programming model, and a translation to other programming models is not trivial. Consequently, analyses relying on the vector clock exchange remain model-specific. This paper proposes an abstraction layer for on-the-fly vector clock exchanges for distributed-memory programs. Based on the programming models MPI, OpenSHMEM, and GASPI, we define common synchronization primitives and explain how model-specific procedures map to our model-agnostic abstraction layer. The exchange model is general enough also to support synchronization concepts of other parallel programming models. We present our implementation of the vector clock abstraction layer based on the Generic Tool Infrastructure with translators for MPI and OpenSHMEM. In an overhead study using the SPEC MPI 2007 benchmarks, the slowdown of the implemented vector clock exchange ranges from 1.1x to 12.6x for runs with up to 768 processes.**

***Index Terms*—Vector clock, distributed memory, synchronization, correctness tools, MPI, OpenSHMEM, GASPI**

I. Introduction

Capturing the ordering of events in parallel program execution is an essential requirement for many correctness analyses. It enables data race detection in shared memory [1] and PGAS programs [2] or computation of alternative message matchings in MPI applications with non-deterministic behavior [3]. To record the causality of events, logical timestamps called *vector clocks* [4], [5] are assigned to them. Subsequently, based on the vector clock comparison of any two events, their happened-before relation can be determined: An event happened before or after the other event, or their execution is not ordered, i.e., they are concurrent. Depending on the application area, some approaches use vector clocks in a postmortem analysis on a program trace to reconstruct the ordering [2], [6], while others use them to verify correctness properties on-the-fly [1].

To analyze the happened-before relation with vector clocks, a tool has to understand the different synchronization concepts of a programming model, e.g., barriers, locks, message exchanges, or collective data exchanges. Hence, tool developers have to study the programming model standard and implement a vector clock exchange accordingly. However, distributed-memory programming models like MPI [7], OpenSHMEM [8], and GASPI [9] have involved synchronization constructs and semantics. For example, MPI has different flavors of collective operations, e.g., reduce, broadcast, or barrier, having different synchronization behavior. Previous vector clock models as in [4] and [5] only consider message exchanges or are specific to synchronization via resources (e.g., locks, mutexes) in shared-memory programs as in [10]. There is no common abstraction how synchronization constructs in distributed-memory programs affect the vector clocks and thus the happened-before order. The missing abstraction leads to tool analyses tailored to a single programming model but not general enough to support other programming models. This is, in particular, the case for hybrid models, e.g., the combined usage of MPI with PGAS models like OpenSHMEM or with shared-memory models like OpenMP [11].

A generic abstraction layer to exchange vector clocks helps to (1) make correctness analyses more independent of the programming model and (2) support new programming models by only requiring translation to the abstraction layer. This paper presents such a generic on-the-fly vector clock exchange model and implementation. We make the following contributions:

- We propose a classification of synchronization concepts in distributed-memory programming models, particularly for MPI, OpenSHMEM, and GASPI.

 (DOI: 10.1109/IPDPSW55747.2022.00093)

- We define a generic on-the-fly exchange model for vector clocks that supports the aforementioned but also other distributed-memory programming models.
- We present a scalable implementation of the clock exchange model on the Generic Tool Infrastructure [12] with translators for MPI and OpenSHMEM and ready to be extended by translators for other programming models.
- An experimental evaluation of our implementation on the SPEC MPI 2007 benchmarks shows slowdown factors of 1.1x to 12.6x for runs with up to 768 processes.

The paper is structured as follows: In Section II, we present background information on vector clocks and the synchronization constructs of MPI, OpenSHMEM, and GASPI. In Section III, we present a classification of synchronization concepts in distributed-memory programs and discuss our general on-the-fly vector clock exchange model. Section IV covers the implementation using the Generic Tool Infrastructure which is evaluated in Section V. Finally, we discuss related work in Section VI and conclude the paper in Section VII.

II. Background

In this section, we give an introduction to the concept of vector clocks and cover important synchronization concepts of the programming models MPI, OpenSHMEM, and GASPI.

A. Vector Clocks

The happened-before relation of events in a parallel program execution can be represented with vector clocks. As described in [5], a distributed system consisting of N single-threaded processes $P := \{P_0, \dots, P_{N-1}\}$ is assumed. Further, each process P_i performs a number of events: Events can either be (1) *internal events* that are only of relevance to the local process executing it, (2) *send events* being the sending of a message to another process, or (3) *receive events* being the reception of a message corresponding to a send event of another process. The events of P_i are enumerated in the order of their occurrence in the set $E_i = \{e_0^i, e_1^i, e_2^i, \dots\}$. The set $E = E_0 \cup \dots \cup E_{N-1}$ contains all events of the distributed computation. The *happened-before order* as defined by [5] is a partial order that formalizes the causality of events in a parallel execution.

Definition 1 (adapted from [5]). *The happened-before order $\xrightarrow{hb} \subseteq E \times E$ is the smallest transitive relation satisfying the following conditions:*

(1) If $e_j^i, e_k^i \in E_i$ occur in the same process P_i and $j < k$, then $e_j^i \xrightarrow{hb} e_k^i$.
(2) If $s \in E_i$ is a send event and $r \in E_j$ is the corresponding receive event, then $s \xrightarrow{hb} r$.

We call two events $a, b \in E$ *concurrent* and write $a||b$, if $a \not\xrightarrow{hb} b$ and $b \not\xrightarrow{hb} a$, i.e., there is no ordering guaranteed between the events.

To record the happened-before order in a distributed system, *logical clocks* can be used: Lamport clocks [13] assign single integer timestamps $C(e) \in \mathbb{N}$ to each event $e \in E$. Such clocks are sufficient for the condition that for any events $a, b \in E$, if $a \xrightarrow{hb} b$, then $C(a) < C(b)$, but the reverse does not hold. Fidge [4] and Schwarz [5] independently proposed to extend the Lamport clocks to *vector clocks*. Instead of only storing a single integer per event, a vector timestamp $V(e) \in \mathbb{N}^N$ is used. Each value $V(e)[i]$ represents the clock value for process i that is stored at event e. The vector clock exchange according to [5] is defined as follows:

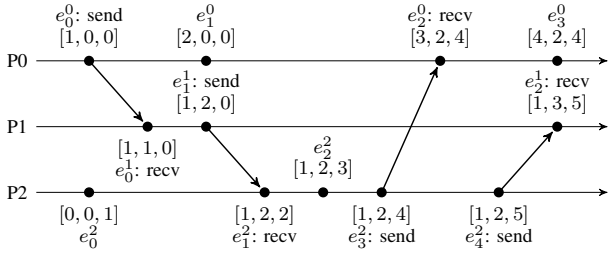


Fig. 1. Vector clock exchange example with three processes using point-to-point communication via messages. The message exchanges are depicted with arrows. For each event e_j^i at process i, the current clock $V_i = V(e_j^i)$ is annotated in the figure.

Definition 2 (adapted from [5]). *Let $P_0, \dots, P_{N-1}$ denote the processes of a distributed computation. The vector clock V_i of process P_i is maintained according to the following rules:*

(1) Initially, $V_i[k] := 0$ for $k = 0, \dots, N-1$.
(2) On each internal event e, process P_i increments V_i as follows: $V_i[i] := V_i[i] + 1$.
(3) On sending message m, P_i updates V_i as in (2) and attaches the new vector to m.
(4) On receiving a message m with attached vector clock $V(m)$, P_i increments V_i as in (2) and updates its current V_i as follows: $V_i := \max\{V_i, V(m)\}$.

Figure 1 illustrates the vector clock exchange corresponding to that definition. When an event e_j^i on process i is encountered, the clock V_i is updated correspondingly and the resulting updated clock V_i is stored as clock of the event $V(e_j^i)$.

Intuitively, an entry $V(e_j^i)[k]$ contains the timestamp of the latest event of process k that is guaranteed to happen before e_j^i. The following theorem proven in [5] explains the relation between vector clocks and the happened-before order:

Theorem 1 (see [5]). *For two events e and e' of a distributed computation, we have*

(1) $e \xrightarrow{hb} e'$ iff $V(e) < V(e')$
(2) $e||e'$ iff $V(e) \not< V(e')$ and $V(e') \not< V(e)$

The comparison of vector clocks is defined as follows:

Definition 3 (see [5]). *Let V, V' be vector clocks of dimension N.*

(1) $V \leq V'$ iff $V[i] \leq V'[i]$ for $i = 0, \dots, N-1$
(2) $V < V'$ iff $V \leq V'$ and $V \neq V'$

In Figure 1, event e_0^0 happened before event e_1^2, since $[1,0,0] < [1,2,2]$. Event e_1^0 is concurrent to event e_1^2, since $[2,0,0] \not< [1,2,2]$ and $[1,2,2] \not< [2,0,0]$.

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093

B. Distributed-Memory Programming Models

In this paper, we consider the three library-based approaches MPI [7], OpenSHMEM [8], and GASPI [9] for distributed-memory programming. While MPI predominantly uses message passing for communication, OpenSHMEM and GASPI use a partitioned global address space (PGAS) model. They all use an SPMD programming style and support C/C++ and FORTRAN programs. We use the following terms to characterize the semantics of a procedure:

- A procedure is *local* if its completion only depends on the calling process, and *non-local*, if completion may depend on the execution of a procedure on another process.
- A procedure is *blocking* if it only returns when the underlying operation is completed. A procedure is *non-blocking* if it does not wait for completion. The user has to explicitly invoke another *completion procedure* to wait for the completion of a non-blocking operation.

The definitions are based on the MPI specification [7, 2.4], but also apply to OpenSHMEM and GASPI. We shortly present basic procedures that provide process synchronization:

1) MPI: The Message Passing Interface (MPI) [7] is the de-facto standard library interface specification for message-passing communication in distributed systems. It supports point-to-point communication between pairs of processes, collective communication between groups of processes, and one-sided communication for remote memory accesses.

Point-to-point communication between pairs of processes is realized through send and receive calls. The standard `MPI_Recv` call is non-local because it will wait for the sending process to call a matching `MPI_Send`. The non-blocking variant `MPI_Irecv` is local; it does not wait for completion. The user has to explicitly call `MPI_Wait` which is then a blocking non-local completion call. For send calls, the synchronization behavior depends on the send mode: A *synchronous* send waits for the matching receive call and is non-local, while a *buffered* send may only buffer the message and is local. The different flavors of receive and send calls have a significant influence on the process synchronization semantics, as we will discuss in Section III.

MPI provides group data exchanges via *collective communication* where the groups are identified using *communicators*. While some procedures explicitly synchronize processes (`MPI_Barrier`), others synchronize processes through the data dependencies of the communication, e.g., for `MPI_Reduce`, the root process has to wait for all other processes to also arrive at the call.

The distinction of a call being local or non-local depends on the role of the process in the communication operation: The `MPI_Reduce` call is non-local for the root process because it has to wait for the other processes to also call it. It is local for all other processes because they only have to send out data to the root and can continue.

In addition to point-to-point and collective communication, MPI provides *one-sided communication*, also called MPI RMA. It allows processes to directly put data to or get data from an exposed part of the remote process' memory using the procedures `MPI_Put` and `MPI_Get` while the target process stays passive. Processes in MPI RMA are synchronized either through passive target synchronization via `MPI_Win_lock` and `MPI_Win_unlock` or through active target synchronization via fences. They also might poll for changes in memory locations.

2) OpenSHMEM: OpenSHMEM [8] provides a PGAS API for distributed-memory systems. Remotely accessible memory regions are allocated as *symmetric data objects* on all processes, either as static variables or on a symmetric heap with the blocking non-local function `shmem_malloc`.

OpenSHMEM does not define any point-to-point communication calls. For collective synchronization, it provides `shmem_barrier_all` and `shmem_sync` for explicit synchronization between processes. Further, it provides different collective communication calls.

Similar to MPI RMA, remote memory in OpenSHMEM can be accessed via non-blocking put and get procedures. Process synchronization is achieved using the call `shmem_wait_until` or similar flavors, which block until a value of a symmetric data object is changed to a provided value.

3) GASPI: The Global Address Space Programming Interface (GASPI) [9] also provides a PGAS API.

Point-to-point communication in GASPI is done with the calls `gaspi_passive_send` and `gaspi_passive_receive`, both functions are blocking and non-local, in terms of synchronization they are similar to `MPI_Ssend` and `MPI_Recv`.

Collective synchronization is achieved via `gaspi_barrier`. The procedure `gaspi_allreduce` performs a reduction operation and broadcasts the result to all processes in a group.

One-sided communication is the main focus of GASPI. *Memory segments* have to be exposed as part of the global address space that can be used in RMA operations. The access to the segments is possible using the non-blocking functions `gaspi_read` and `gaspi_write`. Ensuring causality between the RMA operations is achieved with a *notification* mechanism: Using the procedure `gaspi_notify`, a process can inform the remote process that an RMA operation is completed. The remote process may wait for a notification using `gaspi_notify_waitsome` to assure that RMA operations issued before the notification have been completed.

III. Vector Clock Exchange Model

The original vector clock exchange models described in [4] and [5] assume that the distributed system communicates and synchronizes via messages only. While this is a valid abstraction for point-to-point communication, it does not consider collective synchronization like barriers or synchronization via locks or polling. In this section, we classify synchronization concepts in distributed-memory programs. Based on that, we present our vector clock exchange model that (1) is general enough to capture process synchronization semantics of different programming models, in particular MPI, OpenSHMEM, and GASPI, and (2) is suited as on-the-fly analysis in parallel program executions in terms of efficiency. Preliminary work on this model and has been done in [14].

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093

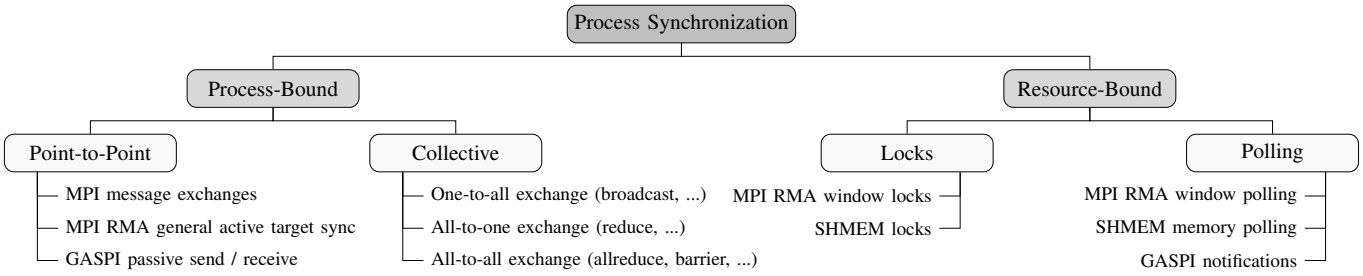


Fig. 2. Classification of synchronization concepts present in MPI, OpenSHMEM, and GASPI.

A. Synchronization Classification

First, we classify the main synchronization concepts in MPI, OpenSHMEM, and GASPI. Figure 2 shows an overview of the classification. The first significant difference between synchronization constructs is whether a process is aware of the other process (or group of processes) it is synchronizing with. In point-to-point communication via message passing, the sending process knows the receiver of a message. Similarly, the receiving process knows the sender at the latest with the reception of the message. The same is true for collective communication, where each process is aware of the group of processes it is synchronizing with. We name this kind of synchronization *process-bound*. For one-sided communication, synchronization is decoupled from communication: Processes synchronize through resources, e.g., locks or via polling on memory addresses. They only know the resource and are not aware of the other process they are synchronizing with. We name this kind of synchronization *resource-bound*.

Process-bound synchronization can be further subdivided into *point-to-point synchronization* which incorporates all synchronization constructs involving exactly two processes, e.g., `MPI_Send` and `MPI_Recv` pairs, and into *collective synchronization* subsuming all synchronization constructs involving arbitrary-sized process groups. For collective synchronization, we differentiate between one-to-all, all-to-one, and all-to-all exchanges, similar to the categorization in the MPI standard [7, 6.2.2], providing different synchronization guarantees. Synchronization constructs might either be *explicit*, i.e., synchronization is their primary purpose as it is the case for a barrier, or *implicit*, i.e., synchronization is instead a side effect than the main intention as, for example, in a reduction.

For resource-bound synchronization, we identified *locks* via MPI RMA and OpenSHMEM. If a process is currently holding a lock that another process also wants to acquire, then it has to wait for that process to release the lock, and therefore the processes synchronize. Further, *polling*, i.e., busy waiting on the state change of a resource, is present in all three programming models. Explicit examples include OpenSHMEM's *shmem_wait_until* calls as well as GASPI's notification mechanism and polling on windows in MPI RMA.

B. System Model

Similar to [5] and described in Section II-A, we assume a distributed system consisting of N single-threaded processes $P := \{P_0, \dots, P_{N-1}\}$ with the given enumeration of events $E_i = \{e_1^i, e_2^i, e_3^i, \dots\}$ for each process P_i. The original system was assumed to communicate via message passing with *send* and *receive* events. Additionally, we assume that the system may (1) use collective communication and (2) exchange data and synchronize using remote memory accesses.

To incorporate collective and resource-bound synchronization into the system model, we generalize the notion of *send* and *receive* events to *signal* and *wait* events that use a *synchronization identifier* $\mathcal{I}$. For process-bound synchronization, $\mathcal{I}$ is the pair of processes (P_i, P_j) that synchronize. Collective synchronization is mapped to multiple *signal* and *wait* events as we will discuss later. In the case of resource-bound synchronization, the synchronization identifier $\mathcal{I}$ is the resource itself. With that, we generalize the happened-before relation as follows:

Definition 4. *The happened-before order* $\xrightarrow{hb} \subseteq E \times E$ *is the smallest transitive relation satisfying the following conditions:*

(1) If $e_j^i, e_k^i \in E_i$ *occur in the same process* P_i *and* $j < k$*, then* $e_j^i \xrightarrow{hb} e_k^i$.

(2) If $s \in E_i$ *is a* ***signal event*** *and* $w \in E_j$ *is the* ***matching wait event with the same synchronization identifier*** $\mathcal{I}$*, then* $s \xrightarrow{hb} w$.

For example, a *send* event s from process P_i to process P_j in the original model is now represented as *signal* event with identifier $\mathcal{I}_s = (P_i, P_j)$ and the matching *receive* event r at process P_j is represented as *wait* event with the same identifier $\mathcal{I}_r = (P_i, P_j)$.

C. Generalized Clock Exchange

We now define the vector clock exchange for the abstracted *signal* and *wait* events: In the original Definition 2 presented in Section II-A, the vector clock is piggybacked to the message exchanged between *send* and *receive* events. This is, however, not possible for resource-bound synchronization because the signaling process is not aware of the waiting process. Thus, we define the process $D(\mathcal{I}) \in P$ as *deposition* of the synchronization identifier $\mathcal{I}$. It identifies the process at which the vector clock is stored for the exchange. The rules for *signal* and *wait* are as follows:

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093

Definition 5 (adapted from [5]). *Let $P_0, \ldots, P_{N-1}$ denote the processes of a distributed computation. The vector clock V_i of process P_i is maintained according to the following rules:*

(1) Initially, $V_i[k] := 0$ for $k = 0, \ldots, N-1$.
(2) On each internal event e, process P_i increments V_i as follows: $V_i[i] := V_i[i] + 1$.
(3) On a signal event s, P_i updates V_i as in (2) and deposits its current vector V_i at $D(\mathcal{I})$. We also say P_i signals to $\mathcal{I}$ or write $P \xrightarrow{s} \mathcal{I}$.
(4) On a wait event w, P_i updates V_i as in (2) and waits for a matching vector $V(w)$ to be deposited at $D(\mathcal{I})$, fetches it, and updates its current V_i as follows: $V_i := \max\{V_i, V(w)\}$. We also say P_i waits for $\mathcal{I}$ or write $P \xleftarrow{w} \mathcal{I}$.

With that generalized clock exchange, we can capture any synchronization construct in MPI, OpenSHMEM, and GASPI.

It is up to the implementation how the deposition $D(\mathcal{I})$ is realized. For point-to-point synchronization, the sending process might deposit its clock at the receiving process using piggybacking or store the clock at a dedicated process such that the receiving process can fetch it from there later on. For resource-bound synchronization, the signaling process might deposit the vector clock at the owning process of a resource, and the waiting process might retrieve it from there.

D. Handling Non-Blocking Operations

Non-blocking synchronization operations comprise two events, the initiation event e_{init} and the completion event e_{comp}, e.g., `MPI_Isend` and the completing `MPI_Wait`. Hence, the underlying operation (and thus synchronization) might occur in a time frame between e_{init} and e_{comp}. This leaves room for interpretation when the *signal* and *wait* events and which vector clock should be sent. Our assumptions are:

1) If the underlying operation is mapped to a *signal*, the *signal* is issued at e_{init}.
2) If the underlying operation is mapped to a *wait*, the *wait* is issued at e_{comp}.

For an operation corresponding to a *signal*, we take and send the vector clock at the earliest possible point in time as a synchronization point. For an operation corresponding to a *wait*, we retrieve and merge the vector clock at the latest possible point in time.

E. Point-to-Point Synchronization

Point-to-point synchronization directly translates to *signal* and *wait* events as shown in Table I. The synchronization identifier is $\mathcal{I} = (P_s, P_w)$ for both the signaling process P_s and the waiting process P_w.

For MPI, the `MPI_Send` and `MPI_Recv` procedures can be mapped to *signal* and *wait* depending on their flavor, we denote the sending process with P_s and the receiving process with P_r in the following. For the *send* procedures, we have to distinguish between the send modes since they have different synchronization semantics: A buffered send `MPI_Bsend` can be mapped to a plain *signal* with $\mathcal{I} = (P_s, P_r)$. In case of a synchronous send `MPI_Ssend`, a process additionally waits for the receiving process, i.e., it establishes bidirectional synchronization with the matching `MPI_Recv`. Thus, the call to `MPI_Ssend` is mapped to a *signal* with $\mathcal{I} = (P_s, P_r)$ and to a *wait* with $\mathcal{I} = (P_r, P_s)$. Similarly, the matching `MPI_Recv` call to a synchronous send is mapped to a wait event with $\mathcal{I} = (P_s, P_r)$ and a signal event with $\mathcal{I} = (P_r, P_s)$. Therefore, in Table I, `MPI_Ssend` and `MPI_Recv` both appear as *signal* and *wait* events.

TABLE I
MAPPING OF POINT-TO-POINT PROCEDURES TO SIGNAL AND WAIT.

Primitive	Procedures
signal	MPI_Send, MPI_(S|B|I|R)send, MPI_I(s|b|r)send, MPI_Wait(_all|_any|_some)^S, MPI_Recv^S, MPI_Sendrecv, MPI_Win_complete
wait	MPI_Recv, MPI_Ssend, MPI_Sendrecv, MPI_Wait(_all|_any|_some), MPI_Win_wait
signal	gaspi_passive_send, gaspi_passive_receive
wait	gaspi_passive_send, gaspi_passive_receive

Procedures marked with S are only mapped if they are part of synchronous communication.

Non-blocking procedures are treated as described in Section III-D, e.g., `MPI_Isend` is mapped to a *signal*, whereas `MPI_Irecv` is mapped to no primitive, instead the corresponding `MPI_Wait` guaranteeing completion is mapped to the *wait*.

In case of GASPI, the procedures `gaspi_passive_send` and `gaspi_passive_receive` are both blocking and the sending is synchronous, therefore both calls are mapped to *signal* and *wait*. OpenSHMEM does not have any procedures with point-to-point synchronization.

Some MPI calls like `MPI_Waitall` or `MPI_Win_wait` might wait on multiple signals from processes. Such procedures can be mapped to multiple *wait* resp. *signal* events. Further, for some procedures, the synchronization identifier can only be determined when the underlying operation is completed, e.g., for an MPI wildcard receive or for `MPI_Wait_any`.

F. Collective Synchronization

Collective operations are called by a designated process group. Such synchronization might be explicit, like barriers, or implicit, through data dependencies like reductions or broadcasts. We characterized all collective operations in the three primitives *all-to-one*, *one-to-all* and *all-to-all* as shown in Table II. Every primitive can be mapped to a corresponding *signal* and *wait* exchange between processes, and we define them accordingly in the following.

The primitive *all-to-one* captures constructs where all processes send data to a designated root process, i.e., the root process waits for a *signal* of all other processes.

Definition 6 (all-to-one). *Let $G \subseteq P$ be the group of processes participating in the operation and $P_r \in G$ the root process. Then $P_r \xleftarrow{w} (P_i, P_r)$ and $P_i \xrightarrow{s} (P_i, P_r)$ for all $P_i \in G \backslash \{P_r\}$.*

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093

TABLE II
MAPPING OF COLLECTIVE PROCEDURES

Primitive	Procedures
all-to-one	MPI_Gather(v), MPI_Reduce
one-to-all	MPI_Bcast, MPI_Scatter(v)
all-to-all	MPI_Barrier, MPI_Alltoall(v|w), MPI_Allgather(v), MPI_Allreduce, MPI_Reduce_scatter(_block), MPI_Win_fence, MPI_Win_free
one-to-all	shmem_broadcast
all-to-all	shmem_barrier_all, shmem_alltoall, shmem_(f)collect, shmem_team_sync, shmem_sync_all, shmem_malloc, shmem_align, shmem_free, shmem_realloc
all-to-all	gaspi_barrier, gaspi_allreduce(_user), gaspi_group_commit, gaspi_segment_(create|use)

For collective operations where a single process sends data to all other processes like in a broadcast, we define *one-to-all*:

Definition 7 (one-to-all)**.** *Let $G \subseteq P$ be the group of processes participating in the operation and $P_r \in G$ the root process. Then $P_r \xrightarrow{s} (P_r, P_i)$ and $P_i \xleftarrow{w} (P_r, P_i)$ for all $P_i \in G \backslash \{P_r\}$.*

The primitive *all-to-all* covers procedures where all processes of a group send and receive data from all other processes:

Definition 8 (all-to-all)**.** *Let $G \subseteq P$ be the group of processes participating in the operation. Then $P_i \xrightarrow{s} (P_i, P_j)$ and $P_i \xleftarrow{w} (P_j, P_i)$ for all $P_i, P_j \in G$ with $P_i \neq P_j$.*

The exchange for the primitives *all-to-one* and *one-to-all* can be implemented with a reduction and broadcast of the vector clocks, respectively. The exchange for the primitive *all-to-all* can be implemented with any *all-reduce* scheme, e.g., maximum reduction of the vector clocks to a root process and broadcast back to all other processes. To avoid deadlocks, the implementation should ensure non-blocking behavior of the reduce, broadcast, and all-reduce scheme.

For MPI collective calls with varying counts of data, e.g., `MPI_Allgatherv`, some send counts or receive counts might be 0, i.e., some processes do not participate in the communication and consequently do not send a *signal* or do not *wait* for a process. While in general supported by the model, in our implementation we assume their synchronization behavior to be identical to their non-varying pendant, e.g., `MPI_Allgather`.

G. Optimizations for Collective Exchanges

As an optimization for the *all-to-all* primitive, an *all-gather* scheme is sufficient to exchange the vector clock entries of the processes P_i in the group G: By definition of the vector clocks, the own local vector clock entry $V_i[i]$ of a process P_i must always be larger or equal to the entry $V_j[i]$ on any other process P_j, i.e.,

$$V_i[i] \geq V_j[i] \text{ for all } P_i, P_j \in P.$$

Thus, instead of each process P_i exchanging the whole vector clock V_i with the other processes, it is sufficient to send its own local vector clock entry $V_i[i]$. In the first *all-to-all* operation in Figure 3, the resulting entry $V_j[i]$ for every process P_j is equal to the local entry $V_i[i]$ of process P_i before the exchange, so it is enough for each process P_i to send the entry $V_i[i]$.

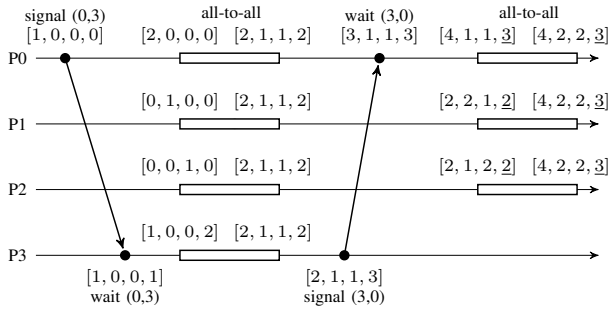


Fig. 3. Vector clock exchange example for collective synchronization. For the *all-to-all* events, the clock before the exchange (after the local increment) and after the exchange is shown. In the second *all-to-all* exchange, only $G = \{P_0, P_1, P_2\}$ synchronize. Although P_3 is not part of the exchange, its value (underscored) also has to be exchanged between processes, otherwise P_1 and P_2 miss the synchronization with P_3. Adapted from [14].

The discussed *all-gather* optimization only applies to the exchange of the clock entries in process group G. If the process group G does not cover all running processes, then in addition the vector clock values of the processes that do *not* participate in the operation have to be exchanged and reduced in an *all-reduce* scheme to account for transitive synchronization. Therefore, the optimized exchange could be interpreted as sparse *all-reduce* operation, where participating processes P_i only send their own clock value $V_i[i]$ and the values $V_i[j]$ of non-participating processes $P_j \in P \setminus G$. Figure 3 illustrates the collective exchange on a subgroup for the second *all-to-all* exchange. It would not be enough that P_0, P_1, and P_2 send only their own local vector clock values. They would also have to exchange the value $V_i[3]$ to reflect the synchronization with the non-participating process P_3.

It is *not* possible to optimize the *all-to-one* primitive similarly by using a *gather* scheme instead of a *reduce* scheme. The difference is that in the *all-to-one* primitive, only the root process P_r has to wait for the signals of the other processes $P_i \in G \setminus \{P_r\}$. The other processes can continue as soon as they have sent their signal. Therefore, an arbitrary process $P_i \in G$ could get into the *all-to-one* primitive and send its vector clock V_i to P_r. Then it could continue with the execution and synchronize (e.g., via point-to-point) with another process $P_j \in G$ *before* P_j encounters the same *all-to-one* primitive and sends its vector clock to the root process P_r. P_j would then get a newer vector clock entry $V_j[i]$ for process P_i and it would not be sufficient for P_j to just send its local entry $V_j[j]$ to P_r, also the newer entry $V_j[i]$ would have to be sent to reflect that P_i has synchronized with P_j in between. To avoid that situation, processes always send the whole vector clock in case of the *all-to-one* primitive.

H. Locks and Polling

Resource-based synchronization through locks and polling can also be mapped to *signal* and *wait* events, where the syn-

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093

TABLE III
MAPPING OF SYNCHRONIZATION THROUGH LOCKS

Primitive	Procedures
signal	MPI_Win_unlock, MPI_Win_unlock_all
wait	MPI_Win_lock, MPI_Win_lock_all
signal	shmem_clear_lock
wait	shmem_set_lock

chronization identifier $\mathcal{I}$ is the resource like a lock identifier or a memory location. The deposition $D(\mathcal{I})$ is a designated process where a signaling process stores its vector and where a waiting process can fetch the vector from. Depending on the implementation, the deposition is the process that is the owner of the resource, but it could also be any other process. Figure 4 illustrates such a vector clock exchange with a resource.

Table III shows how we map locking to *signal* and *wait*. Intuitively, a lock call to a resource maps to a *wait*, because a process has to wait for another process to unlock the resource (except the initial lock where we assume that the vector clock is initially zero in all entries). An unlock call maps to a *signal* because that process signals to the process that waits for the resource. While this is true for locking via SHMEM, the locking synchronization via MPI RMA is weaker: According to the MPI standard [7, 11.5.3], on a call to `MPI_Win_lock`, the MPI implementation might delay the actual locking until `MPI_Win_unlock` is called, but not if a process locks its window locally. Nevertheless, we assume here as simplification that `MPI_Win_lock` is always mapped to *wait* and `MPI_Win_unlock` to *signal*.

As shown in Table IV, all three programming models permit synchronization via polling on memory locations (MPI RMA and SHMEM) or notifications (GASPI). In MPI RMA, the waiting process can poll on a local memory location without calling into the MPI library. Any writing MPI RMA access from remote could be the *signal*. For OpenSHMEM, the call `shmem_wait_until` waits for a state change of a given memory location, while the signal might take place through any SHMEM RMA memory modification from remote.

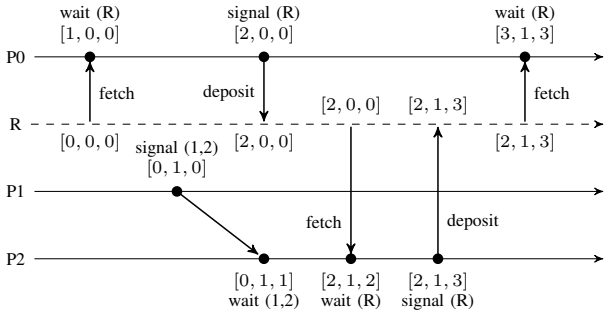


Fig. 4. Vector clock exchange example for resource-based synchronization on a resource R located on an arbitrary other process. Processes P_0 and P_2 synchronize via resource R by depositing the clock at and fetching it from that resource. The initial vector clock of the resource is $[0, 0, 0]$.

TABLE IV
MAPPING OF SYNCHRONIZATION THROUGH POLLING

Primitive	Procedures / Actions
signal	(potentially) any MPI RMA write
wait	(potentially) any local memory read
signal	(potentially) any SHMEM RMA write
wait	shmem_wait_until(_any|_all|_some)
signal	gaspi_notify, gaspi_write_notify
wait	gaspi_notify_waitsome

Since exchanging vector clocks with every (remote) memory access is infeasible in practice, we rely on user annotations in code for this kind of synchronization in our implementation as we discuss in Section IV-F.

I. Limitations

Synchronization through tasking and task dependencies have been left out in the model, because none of the three considered programming paradigms (MPI, OpenSHMEM, GASPI) supports them. However, it could be modeled as a subclass of resource-bound synchronization where task dependencies or the tasks itself are considered as resources. Further, we left out `MPI_Scan` and its flavors since its data dependencies are complex and an application should not rely on its implicit synchronization behavior.

The model is currently limited to single-threaded distributed-memory programs. An extension to multi-threaded paradigms as MPI+OpenMP would require to (1) define a mapping from the concrete shared-memory synchronization concepts to our introduced model primitives and (2) increase the size of vector clock to incorporate the synchronization between threads. In case of MPI, processes itself are not aware of thread synchronization, but only can synchronize with other processes. Therefore, it might be beneficial to manage a *process vector clock* for the process synchronization exchange and in addition per process a *thread vector clock* for the thread synchronization.

IV. IMPLEMENTATION

We implemented the described vector clock mechanism as part of GTI [12] in two components: First, a generic vector clock exchange module that implements the primitives *signal*, *wait*, *one-to-all*, *all-to-one*, *all-to-all* independent of the programming model. Second, a wrapper specific to the programming model that translates the concrete procedure calls to the abstracted primitives of the generic vector clock exchange module. We implemented wrappers for MPI and OpenSHMEM. A wrapper for GASPI is left out as future work. In the following, we will describe details of the tool architecture and its implementation.

A. GTI

The Generic Tool Infrastructure (GTI) [12] provides an abstracted framework allowing tool developers to design scalable event-based analysis tools for MPI programs. GTI wraps and

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093

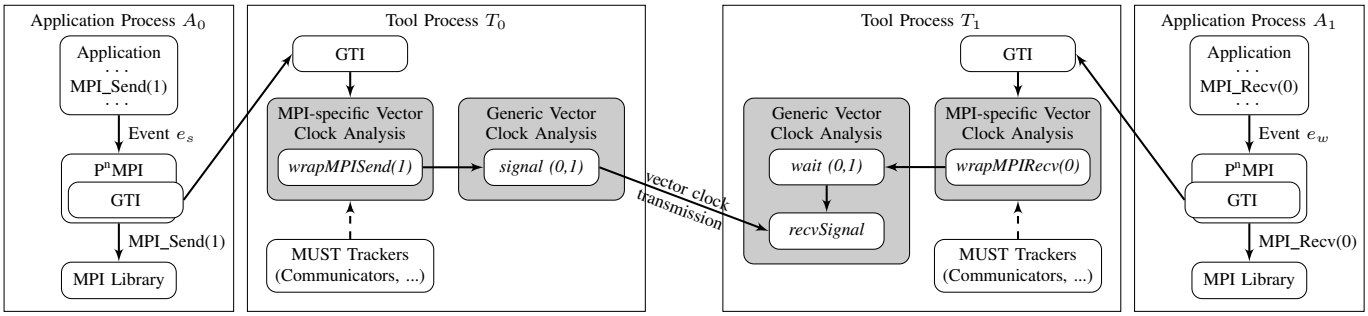


Fig. 5. Vector clock exchange workflow for an MPI message transfer via `MPI_Send` and `MPI_Recv`. The gray parts mark the implemented modules that were added for the vector clock exchange. Adapted from [12] and [14].

intercepts MPI calls using P^nMPI [15], but also supports other wrapping libraries to work with other programming models. A tool based on GTI consists of several analysis modules written in C++ that can be dynamically loaded at runtime. In addition to the application processes, tool processes or tool threads can be used to offload analysis computations. Further, communication channels between application and tool processes can be defined using different communication protocols (MPI, TCP or similar) and communication strategies (blocking or non-blocking send of analysis data).

We add a new analysis module for the vector clock analysis that implements the generic vector clock exchange primitives. We decided to create an additional tool process for each application process that manages the vector clock. This allows us to offload the vector clock exchange independent of the application process.

B. MUST

The Marmot Umpire Scalable Tool (MUST) [16] is a runtime correctness checking tool for MPI applications that utilizes GTI as communication infrastructure. MUST can detect various MPI usage errors, e.g., deadlocks, using incompatible datatypes, passing wrong arguments, and so on. The correctness analyses are defined in separate analysis modules.

MUST provides analysis modules to track communicators and MPI RMA windows during the execution of an MPI program. For the MPI-specific vector clock exchange, we use the communicator tracking module to map MPI communicators to generic process groups and identifiers. The RMA window tracking module is used to understand synchronization behavior for process synchronization calls in MPI RMA.

C. Analysis Workflow

The workflow of the vector clock analysis for a simple MPI message transfer between application processes A_0 and A_1 is depicted in Figure 5. When the application process calls an MPI procedure, it is intercepted by P^{n}MPI and then passed to the tool process using GTI. MPI calls are translated in an MPI-specific wrapper module to the generic vector clock primitives within the tool process. In the concrete example, the `MPI_Send` call is mapped to *signal*, and the `MPI_Recv` call is mapped to *wait*. The generic vector clock analysis then transmits the vector clock from the sending process A_0 to the receiving process A_1 using the communication channel between tool processes T_0 and T_1. If the receiving process A_1 did not call the matching `MPI_Recv` when the signal from T_0 is received at T_1, then the vector clock information is buffered for a later merge. When the analysis of the tool process is done, the application can continue with the execution, i.e., the MPI call is then passed to the MPI library.

The implementation minimizes the waiting times in the application processes as much as possible for an on-the-fly analysis: Sending the vector clock from A_0 is decoupled from the application execution, i.e., after the send call has been wrapped in the tool process T_0, the application process A_0 can continue, even though the vector clock might not have yet been received at tool process T_1. However, the application has to be stalled on the receiving process until the vector clock from the sending process has arrived at T_1 because receiving an outdated vector clock is of limited use for subsequent on-the-fly correctness analyses. For offline analyses, where receiving outdated vector clocks is of no harm, our tool can be configured to exchange the clocks without application stalling.

As described in Section III-D, we map non-blocking operations to vector clocks in a way such that the least possible synchronization is assumed. For non-blocking operations that are mapped to a *wait*, we postpone a vector clock merge until the operation is completed.

For MPI, processes are identified using ranks in combination with communicators. In MUST we use a communicator tracking module to translate all ranks to the world communicator and then pass that to the generic vector clock analysis module. Further, messages might be sent through different queues, so we added a *queue identifier* to *signal* and *wait* transmissions to differentiate between messages with different tags in case of MPI. Other programming models bring their own abstraction of process identifiers and queue identifiers which could also be translated as we do with the MUST tracking modules for MPI.

D. Collectives

As described in Section III-F, the collective synchronization primitives *all-to-one*, *one-to-all*, and *all-to-all* map to *reduce*, *broadcast*, and a sparse *all-reduce* operation on the corre-

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093

Process A	Process B
	window location X (initialized with 0)
MPI_Win_lock(B, EXCLUSIVE)	
AnnotateResourceSignal(X, B)	
MPI_Put(1, X)	while(X == 0)
MPI_Win_unlock(B)	noop()
	AnnotateResourceWait(X, B)

Fig. 6. Annotation of synchronization in MPI RMA via polling in the unified memory model: Process A uses `MPI_Put` to update the window location `X` at process B from 0 to 1. Process B will only leave the `while` loop when the update from process A to window location `X` is visible. Thus, process B synchronizes with the `MPI_Put` operation from process A which is correspondingly annotated.

sponding vector clocks for a given process group, respectively. Considering the optimizations from Section III-G for the *all-to-all* primitive, *all-reduce* can be understood as sparse operation, where participating processes only send their own vector clock value (*all-gather*) and the values of non-participating processes, the remaining values are not transmitted.

GTI by default only provides point-to-point exchanges between processes, so we implemented our own collective operations (broadcast, reduction, gather) using a binomial tree for vector clocks on top of it.

The basic analysis principle for collectives is similar to that shown in Figure 5. To translate MPI collectives, we use the MUST communicator tracking module to derive the process group out of a given communicator.

E. Resource-Bound Synchronization

For synchronization via a resource R, the deposition $D(R)$ identifies the process at which the vector clock of the resource can be found. In our implementation, the deposition is the tool process associated with the application process that owns the resource, e.g., for an exclusive lock on an MPI RMA window, the owning process of the window is used as deposition. If a process signals to a resource, then the local vector clock is sent to the deposition process, where the clock is then buffered locally. The waiting process then has to (1) send the request for the clock to the deposition process and (2) wait for the vector clock to be delivered by the deposition process. Using one-sided communication for depositing and fetching might be beneficial in performance but is currently not supported in the communication layer of GTI.

F. User Annotation Interface

There might be synchronization constructs in a program that cannot be attributed to a procedure of the programming model, as discussed for resource-bound synchronization like polling in Section III-H. Further, users might want to trigger a vector clock exchange manually in their code, e.g., to capture (future) synchronization constructs that are not mapped yet by our implementation. For those cases, our implementation provides a user annotation API to add the intended synchronization. The user can add those annotation calls in the code. In case of MPI, the calls are forwarded via `MPI_Pcontrol` to our tool. If the user runs the code without our tool, those calls have no effect, while a run with our tool will trigger the corresponding vector clock exchanges. The following generic procedures are provided:

- `AnnotateTick()`: Increment local vector clock entry (of the calling process) by one.
- `AnnotateProcessSignal(target, queue)`: Send signal to given `target` process using the given `queue`.
- `AnnotateProcessWait(origin, queue)`: Wait for signal from the given `origin` process on the given `queue` and merge the own vector clock with the received vector clock.
- `AnnotateResourceSignal(resource, deposition)`: Deposit current vector clock to the given `resource` at the given `deposition` process.
- `AnnotateResourceWait(resource, deposition)`: Fetch vector clock from the given `resource` at the given `deposition` process and merge it with the own vector clock.

Figure 6 shows in pseudocode how such annotations work for MPI RMA polling in the unified memory model. The polling mechanism in OpenSHMEM could be annotated similarly.

The presented generic annotation calls are independent of the concrete programming model. Thus, wrapping functionality specific to the programming model like translating ranks to the MPI world communicator would have to be done manually by the user. To avoid that, we additionally provide calls specific to the programming model, e.g., `AnnotateProcessWait(origin, communicator, tag)` to annotate *wait* events in MPI where the `origin` rank is translated to the world communicator.

V. Overhead Evaluation on MPI Applications

We evaluated the runtime overhead of our implemented on-the-fly vector clock analysis using the SPEC MPI 2007 [17] benchmarks. The results are discussed in this section.

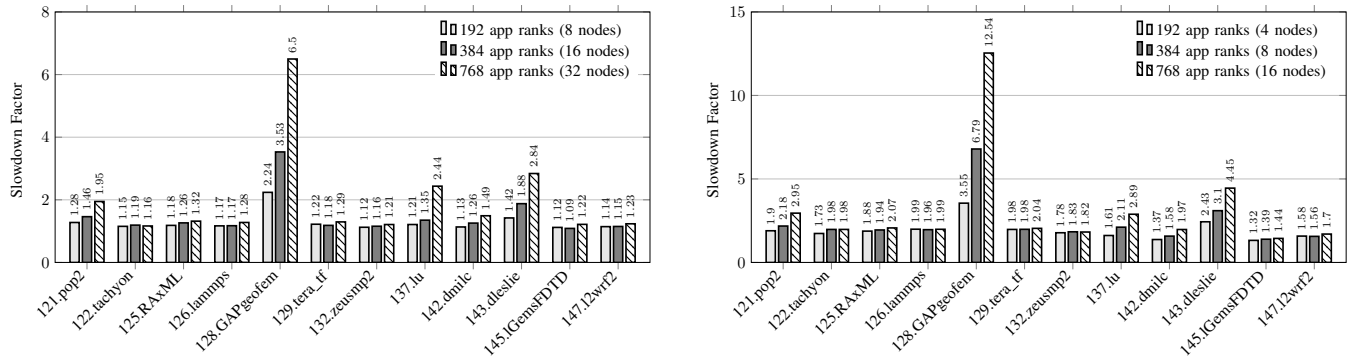
A. Experiment Setup

The measurements were performed on the RWTH CLAIX18 cluster [18] consisting of in total 1200 nodes connected via Intel Omni-Path. Each node is equipped with two Intel Skylake Platinum 8160 processors with a total of 48 physical cores with SMT disabled and 192 GiB of total main memory. We used the Intel MPI library in version 2018.5 and the Intel C/C++ and FORTRAN compiler in version 19.0.

Our experiments were run with up to 32 nodes. Since the vector clock analysis requires an additional tool process to be spawned along with each application process, we analyzed the overhead with two different setups:

1) Split nodes: We split the available cores per node equally into application and tool processes. For the baseline measurement, we use only half of the available physical cores per node for the application processes. For the vector clock measurement, we use the other half of the remaining cores for the tool processes. Thus, the number of application

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093



(a) *Split nodes* configuration (24 app. + 24 tool ranks on 48 cores)

(b) *Shared nodes* configuration (48 app. + 48 tool ranks on 48 cores)

Fig. 7. Overhead evaluation with SPEC MPI 2007 benchmarks

processes stays the same between both measurements, while the number of utilized physical cores used for the vector clock measurement is doubled.

2) Shared nodes: In this setup, we do not use additional cores, but pin each tool process to the same physical core that the corresponding application process is running on. Thus, the baseline setup fully utilizes the number of available cores per node, while the vector clock measurement adds the same number of tool processes to the execution.

We ran all benchmarks of SPEC MPI 2007 [17] in version 2.0 that have a large dataset *lref* with both setups using 192, 384, and 768 application ranks. For the *split node* setup, that means we used 8, 16, and 32 nodes and for the *shared nodes* setup, we used 4, 8, and 16 nodes, respectively. For each benchmark application, the median out of three runs is reported.

B. Results

The benchmark results for both setups are shown in Figure 7. For the *split node* setup, the slowdown factor is for most applications in the range of 1.1x to 2.8x. The only exception is *128.GAPgeofem* with a slowdown factor of 6.5x for 768 ranks. Compared to the other benchmarks, it issues roughly 1,500 calls per second and rank and thus significantly more than the other benchmarks as discussed in [14]. The number of collective MPI calls per second and rank for this benchmark doubles when the number of ranks is doubled. Since each collective call requires a vector clock exchange via our implemented binomial tree reduction, broadcast and all-reduce schemes using the GTI communication infrastructure, the doubled overhead is expected in that case.

For the *shared node* setup, as expected, the slowdown factor is higher compared to the *split node* setup: It ranges from 1.3x to 4.5x for all applications except *128.GAPgeofem* with a slowdown of 12.54x for 768 ranks.

For both setups, doubling the number of application ranks increases the overhead only slightly for most applications. Exceptions are *121.pop2*, *128.GAPgeofem*, *137.lu*, and *143.dleslie* for which the overhead roughly doubles when doubling the number of ranks.

C. Discussion

The considered benchmarks all use point-to-point and collective synchronization. The results show that the expected slowdown factor for most of such applications is around 1.5x for the *split node* setup and around 2.0x for the *shared node setup*. As expected, a high number of issued MPI calls per second significantly increases the overhead, as we can see for *128.GAPgeofem*. Each communication operation in a run with N processes requires a clock exchange of a whole vector (N integers) for point-to-point and up to N vectors (N^2 integers) for collective synchronization. Taking into account that the analysis data has to travel through the tool process infrastructure as shown in Figure 5, we consider the overhead for an on-the-fly analysis as reasonable. Further, due to the asynchronous execution of application and tool processes (where possible) and asynchronous intercommunication of the tool processes, the overhead remains reasonable.

VI. Related Work

Vector clock exchanges have been used for data race detection in different contexts: Shared memory race detectors [1], [10], [19] rely on them to determine the happened-before relationship between events on-the-fly. For detection of races and memory consistency errors in MPI RMA programs, MC-CChecker [2] uses vector clocks in a postmortem analysis to mitigate some false positive cases of its predecessor MC-Checker [20]. The authors define a timestamping system for MPI RMA programs based on encoded vector clocks [21] that can track point-to-point and collective synchronization. Compared to their MPI-specific approach, our clock analysis can be performed at runtime and can also capture resource-bound synchronization.

Besides race detection, vector clocks are used to replay executions: The formal verifier DAMPI [3] uses Lamport and vector clocks to discover all possible message matchings in MPI programs. It records the clocks during the first program execution and then replays the program while enforcing possible alternative message matches based on the clock analysis. The utilized vector clock exchange is similar to our approach

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093

but does not consider resource-based synchronization and is specific to MPI. There are other replay approaches, e.g., timestamping memory accesses using vector clocks [6] to replay them deterministically. However, they are orthogonal to our approach that instead captures the synchronization between processes focusing on on-the-fly correctness checking.

All related vector clock analysis approaches are specific to a particular programming model and do not support multiple programming models. To the best of our knowledge, this is the first approach to define a generalized vector clock exchange for distributed-memory programs.

VII. Conclusion

In this paper, we presented a classification of synchronization concepts in distributed-memory programs and thereby identified the main categories process-bound and resource-bound synchronization. Based on the classification, we defined a generic on-the-fly vector clock exchange model with the primitives *signal* and *wait* for point-to-point synchronization and *all-to-one*, *one-to-all*, and *all-to-all* for collective synchronization. The exchange model supports MPI, OpenSHMEM, and GASPI, but also other distributed-memory programming paradigms.

Our implementation based on the Generic Tool Infrastructure proves the applicability of the generic vector clock exchange. It enables vector clock analyses of MPI and OpenSHMEM programs and is ready to be extended by further programming models. Our evaluation of MPI applications has shown that the overhead of our implementation stays manageable and is ready to be used in correctness analysis tools. Depending on the application's communication scheme and the number of communication calls in relation to computation, the induced overhead may vary, in particular from 1.1x to 12.6x for SPEC MPI 2007 with up to 768 processes. Moreover, we observed that applications extensively using collective synchronization have higher slowdowns than those using predominantly point-to-point communication.

In future work, we will use the generic vector clock implementation to detect data races for remote memory accesses, e.g., *put* and *get* accesses in MPI RMA or OpenSHMEM. Further, we plan to extend the vector clock exchange to also support shared memory programming models like OpenMP such that we can also analyze hybrid MPI+OpenMP programs. This will require an extension of the vector clock exchange mechanism to additionally support threads.

The implementation source code and the evaluation results are available at https://doi.org/10.5281/zenodo.6363136.

Acknowledgements

Parts of this work were done in the master thesis of Felix Tomski [14] under the supervision of Simon Schwitanski.

References

[1] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with LLVM compiler," in *International Conference on Runtime Verification*, pp. 110–114, Springer, 2011.

[2] T.-D. Diep, K. Fürlinger, and N. Thoai, "MC-CChecker: A clock-based approach to detect memory consistency errors in MPI one-sided applications," in *Proceedings of the 25th European MPI Users' Group Meeting, EuroMPI'18, Barcelona, Spain*, ACM, 2018.

[3] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. De Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for MPI programs," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA*, pp. 1–10, IEEE, 2010.

[4] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, pp. 56–66, 1988.

[5] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed Computing*, vol. 7, no. 3, pp. 149–174, 1994.

[6] X. Qian, K. Sen, P. Hargrove, and C. Iancu, "SReplay: Deterministic sub-group replay for one-sided communication," in *Proceedings of the 2016 International Conference on Supercomputing, ICS 2016, Istanbul, Turkey,*, pp. 1–13, 2016.

[7] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.1." http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015. [online; accessed 18-March-2022].

[8] "OpenSHMEM: Application Programming Interface Version 1.4." http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf, 2017. [online; accessed 18-March-2022].

[9] "Gaspi: Global Address Space Programming Interface Version 17.1." https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-17.1.pdf, 2017. [online; accessed 18-March-2022].

[10] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland*, pp. 121–133, ACM, 2009.

[11] OpenMP Architecture Review Board, "OpenMP Application Programming Interface Version 5.2." https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf, 2021. [online; accessed 18-March-2022].

[12] T. Hilbrich, M. S. Müller, B. R. De Supinski, M. Schulz, and W. E. Nagel, "GTI: A generic tools infrastructure for event-based tools in parallel systems," in *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China*, pp. 1364–1375, IEEE, 2012.

[13] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[14] F. Tomski, "Developing an On-the-Fly Vector Clock Exchange for Distributed Memory Systems Using the Generic Tool Infrastructure," Master Thesis, RWTH Aachen University, Aachen, 2021.

[15] M. Schulz and B. R. De Supinski, "P^{N}MPI tools: a whole lot greater than the sum of their parts," in *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, Reno, Nevada, USA*, pp. 30:1–30:10, ACM, 2007.

[16] T. Hilbrich, M. Schulz, B. R. Supinski, and M. S. Müller, "MUST: A scalable approach to runtime error detection in MPI programs," in *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, Dresden, Germany*, pp. 53–66, Springer, 2009.

[17] "SPEC MPI 2007 Benchmark Suite." https://www.spec.org/mpi2007/. [online; accessed 18-March-2022].

[18] "CLAIX18: RWTH Compute Cluster." https://www.itc.rwth-aachen.de/go/id/hisv. [online; accessed 18-March-2022].

[19] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 179–190, 2003.

[20] Z. Chen, J. Dinan, Z. Tang, P. Balaji, H. Zhong, J. Wei, T. Huang, and F. Qin, "MC-Checker: Detecting memory consistency errors in mpi one-sided applications," in *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA*, pp. 499–510, IEEE, 2014.

[21] A. D. Kshemkalyani, A. Khokhar, and M. Shen, "Encoded vector clock: Using primes to characterize causality in distributed systems," in *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018*, pp. 1–8, ACM, 2018.

Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, Matthias S. Müller:
An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs.
2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, pp. 530-540
DOI: 10.1109/IPDPSW55747.2022.00093