Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

Konzepte zur Laufzeiterkennung von kritischem Wettlauf für OpenSHMEM-Programme

On-the-Fly Data Race Detection for OpenSHMEM Programs

Bachelorarbeit

Sven Klotz
Matrikelnummer: 381590

Aachen, den 6. Juni 2023

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (*)
Zweitgutachter:	Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen (†)
Betreuer:	Simon Schwitanski, M.Sc. (*)

(*) Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(†) Lehrstuhl für Softwaremodellierung und Verifikation, RWTH Aachen University

communicated by Prof. Matthias S. Müller

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 6. Juni 2023

Abstract

OpenSHMEM is a distributed-memory programming model part of the *Partitioned Global Address Space* family of programming models. It provides vendors with a standard *Application Programming Interface* to implement SHMEM libraries and facilitates the portability and predictability of OpenSHMEM applications. The OpenSHMEM effort aims to imitate the success of the widely used *Message Passing Interfacing* programming model while focusing exclusively on one-sided communication.

One-sided communication or *Remote Memory Access* enables a process to directly access and modify memory regions of a remote process without actively involving the remote process. Although OpenSHMEM's one-sided communication has many significant desirable properties, it also has drawbacks. Specifically, the one-sided nature of this communication results in applications being prone to contain data races as the remote process is unaware of the remote memory accesses. To prevent this, a developer must ensure proper synchronization between all types of memory accesses, remote or local.

This thesis discusses the semantics of OpenSHMEM communication and synchronization and classifies what constitutes a data race within an OpenSHMEM program. Based on this definition, a semantics-driven search for data races in this thesis results in 35 data race test cases that are classified and labeled as many share common properties. Additionally, this thesis uses the insights gained from semantics-driven search to formalize the notion of data races by adapting an existing semi-formal model of the MPI RMA semantics to OpenSHMEM. Based on this model, this thesis presents the concepts required for on-the-fly data race detection in OpenSHMEM programs and a prototype data race detection tool for OpenSHMEM based on this on-the-fly method.

The approach performs happens-before analysis using vector clocks and uses memory consistency rules to determine the earliest and latest points in logical time when a memory access could occur. The prototype annotates these memory accesses in a shared-memory race detector that performs the actual race detection. A separate race detection covers only special cases. Lastly, this thesis evaluates the prototype using the data race test cases developed for this thesis. The results of this evaluation are promising for our prototype as it achieved an accuracy of 0.86. This shows that the on-the-fly data race detection approach works in practice for OpenSHMEM.

Keywords: HPC, OpenSHMEM, One-Sided Communication, Remote Memory Access, Data Races, Correctness Checking

Contents

List of Figures

List of Tables

List of Codes

1. Introduction

OpenSHMEM is a distributed-memory programming model from the *Partitioned Global Address Space* (PGAS) family and can be used to implement *Single Program Multiple Data* (SPMD) style programs. It aims to imitate the success of the *Message Passing Interface* (MPI) as a library interface specification and provide a standard *Application Programming Interface* (API). This API specification can then be used to implement portable applications between different library implementations and have consistent results [8, Ch. 1-2].

At the core of OpenSHMEM are one-sided communication routines, which, compared to traditional message-passing models, do not require the target to call any OpenSHMEM routines actively. The one-sided nature of communication allows the overlap of it and computation, and in conjunction with modern technologies such as *Remote Direct Memory Access* (RDMA) [11], it can improve communication efficiency in an application.

OpenSHMEM began as a vendor-specific parallel-programming model for Cray Research, Inc. in 1993 under the name *SHMEM* [8, Annex E]. It was a proprietary programming model for Cray and later SGI platforms. However, the initial open and standardized OpenSHMEM specification was only released in 2012, making it a comparatively new programming model. As OpenSHMEM matures, it has gained popularity, and its specification of OpenSHMEM has been expanded upon. Several vendors have developed implementations of OpenSHMEM to aid development efforts of complex software easier and simplify the usage of features on their systems. Some of these expansions are available from vendors such as NVIDIA with NVSHMEM [35] and AMD with ROCm SHMEM [1]. In their implementations, it is possible to use the extensions made in their OpenSHMEM implementations to directly access the memory of their accelerators, which reduces the programming complexity and overhead of distributing workloads running on accelerators across multiple nodes in a system. This active development effort from major vendors indicates that OpenSHMEM will also stay relevant in the future.

While the development of OpenSHMEM is continuing, the surrounding ecosystems and the development of debugging tools need to catch up. At the time of writing and to the best of our knowledge, only two correctness-checking tools exist for the OpenSHMEM programming model. These tools are the OpenSHMEM Analyzer [14] and the OpenSHMEM checker [5], both of which are compiler-based tools. Both of these tools are general correctness tools for OpenSHMEM. However, data races common in parallel applications are some of the most difficult to debug errors. In other programming models, such as MPI RMA, OpenMP, or general shared-memory programming, specialized tools have been developed to detect them. Only

OpenSHMEM Checker claims any, even though experimental and minimal support, for detecting data races.

Thus, this thesis aims to develop a specialized data race detection prototype for OpenSHMEM to study the feasibility of such a tool for OpenSHMEM and set the groundwork for future work on data races in OpenSHMEM. To this end, we will examine the semantics of the OpenSHMEM 1.5 specification, discuss under which conditions data races can occur, and examine previous efforts in other programming models.

The rest of this thesis is structured as follows: Chapter 2 provides a general introduction to OpenSHMEM 1.5. Chapter 3 will discuss data races in OpenSHMEM in detail and propose a data race classification system. This chapter also introduces a set of test cases for data races that can be used to evaluate data race detection tools for OpenSHMEM. Chapter 4 contains some of the main contributions of this thesis. It presents a semi-formal model for OpenSHMEM, adapted from a semi-formal for MPI RMA proposed by Hoefler et al. [17]. This adapted model is used in this chapter to reason about potential data races in an execution of an OpenSHMEM program and, to a limited extent, formalize the notion of data races in OpenSHMEM. Similar to previous work on data race detection in MPI RMA, also based on the model by Hoefler et al., this chapter presents an on-the-fly data race detection model for OpenSHMEM. This race detection model uses a mixture of happens-before and memory consistency analysis to identify potential data races. Chapter 5 contains the other significant contribution of this thesis. It presents the prototype implementation for data race detection in OpenSHMEM and the central components necessary for its race detection approach. In Chapter 6, the test cases from Chapter 3 are used to evaluate the classification quality of the prototype implementation. Lastly, Chapter 7 finishes this thesis by discussing the work and possible future work required to improve the data race detection model and the prototype.

2. PGAS programming model: OpenSHMEM

OpenSHMEM is a *Partitioned Global Address Space* (PGAS) library interface specification. The specification provides a standard *Application Programming Interface* (API) for vendors to implement SHMEM libraries. The standardized interface allows developers to implement *Single Program Multiple Data* (SPMD) style applications that are source code compatible between different library implementations of OpenSHMEM, and the results of all OpenSHMEM operations are consistent and predicatable [8, Ch. 1, Ch. 2].

The central feature of OpenSHMEM is the one-sided nature of its communication operations, allowing the transfer of data between processes without requiring the target process to participate in the communication actively. The specification provides several mechanisms to perform loads and stores on the publicly accessible memory regions of a remote process. A developer can transfer single values, continuous arrays, or strided memory between processes using these mechanisms. In short, the attempt is to recreate the success of the long-standing MPI standard [33] while offering a modern and straightforward programming model as the complexity and scale of HPC algorithms require increasing development effort [54].

2.1. Overview

As the name implies, the OpenSHMEM specification is part of the PGAS programming model family which already contains many languages and libraries. Every member of this family consists of three significant components [3]:

- A set of processing elements, each of which has attached memory. Parts of this memory can be declared *private*, which is not accessible by other processing elements.
- A mechanism by which at least some of the memory can be declared *shared*.
- Every shared memory location has an exposed *affinity* based on which the programmer can implement specific access strategies.

OpenSHMEM implements PGAS in a *Single Program Multiple Data* (SPMD) style where processes, or *Processing Elements* (PE), are spawned based on the same executable and can interact, synchronize or exchange information in diverging

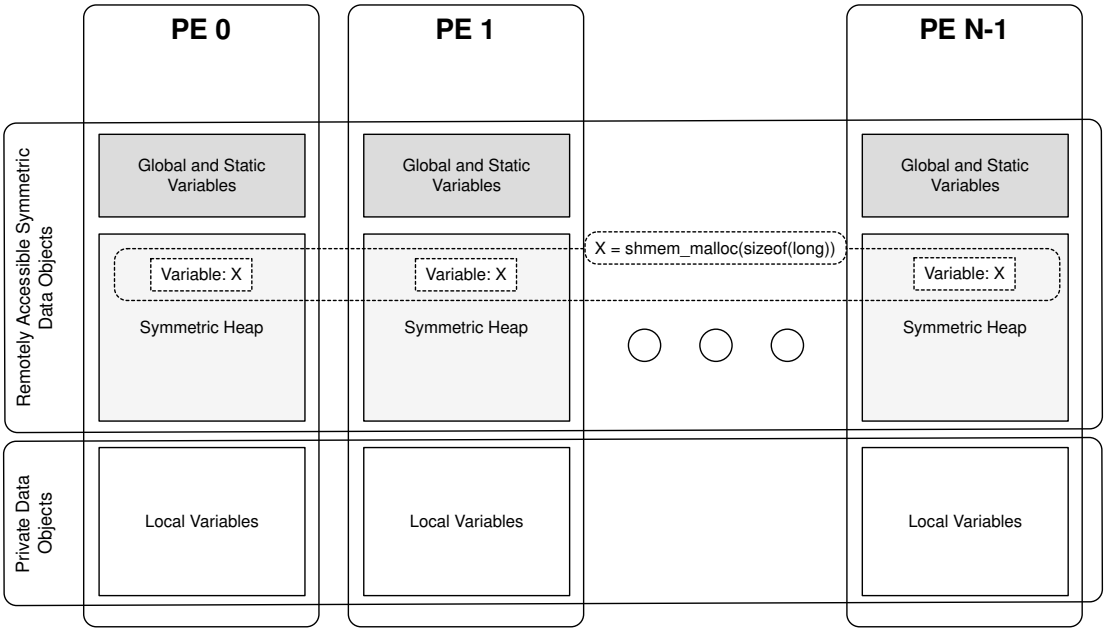


Figure 2.1.: OpenSHMEM Memory Model [8, Ch.3]

branches based on data provided by the library runtime. To this end, each PE has local or *private data objects* and *remotely accessible data objects*. The OpenSHMEM API allows operations on both types of these data objects. However, on private data objects, only operations from the local PE are valid. Therefore remotely accessible data objects act as the mechanism to share information among PEs.

The fundamental design decision behind OpenSHMEM operations is that data transfers are *one-sided* in nature. Therefore a data transfer operation does not require any active participation of the remote PE. This lack of participation from the remote PE implicitly allows for overlap between communication and computation and can hide data transfer latencies, "which makes OpenSHMEM ideal for unstructured, small-to-medium-sized data communication patterns." [8, Ch.2]

2.2. Memory model

As described in Section 2.1, OpenSHMEM defines remotely accessible and private data objects. These data objects are partitioned into two logical memory regions. Understanding the differences and interactions of these regions is essential to understand the data race detection later in this thesis. An illustration of the whole memory layout of OpenSHMEM is shown in Figure 2.1.

Private data objects are uncomplicated in their definition. They are stored in the local memory of each PE. Another PE cannot access them through OpenSHMEM routines, or in short, they behave like the memory in a traditional imperative

programming model. More specifically, private data objects follow the memory model of *C* [8, Ch.3].

In comparison, remotely accessible data objects are more complex. As the name implies, remotely accessible data objects can be accessed by other PEs using a variety of OpenSHMEM routines. In the OpenSHMEM specification, these objects are referenced under the name *Symmetric Data Objects*. Symmetric is a crucial term in this case as each symmetric data object is replicated across all PEs where that object is *accessible* using the same name, type, and size. Therefore these symmetric data objects have the same memory footprint across all PEs.

There are two different kinds of symmetric data objects in OpenSHMEM:

- Variables with *static* storage duration are symmetric data objects as long as they are not defined in a dynamic shared object (DSO). In simpler terms, these are global and static C/C++ variables regardless of their linkage. These variables are present during the entire execution of the program and are initialized only once, before the start of the *main function*. Therefore in an SPMD model, these variables are present on all PEs.

- C and C++ data allocated by the OpenSHMEM implementation using memory management routines are symmetric data objects. These data objects are dynamically allocated during the program's runtime using OpenSHMEM memory management routines like `shmem_malloc`. These allocations happen collectively, ensuring that these dynamic symmetric data objects are also present on all PEs. While OpenSHMEM tries to handle memory management similarly to the standard C API, there are some critical differences.

  Dynamic memory allocations in OpenSHMEM happen on a special memory region called the *Symmetric Heap*. The Symmetric Heap is a memory region created by the implementation during the execution of a program and has a fixed size specified by the `SHMEM_SYMMETRIC_SIZE` environment variable.

A small example of this layout in actual source code is shown in Code 2.1, which provides a few examples of variable declarations and dynamic allocations that are a mixture of symmetric and private data objects.

2.2.1. Addressing symmetric and private data objects

OpenSHMEM routines generally reference data objects through the local pointer to the desired data object. Depending on the type of data object the address is refered to differently. If the data object is a private data object, then the address is called a *local address*. This local address is valid for direct memory access on the PE and behaves as expected from the C/C++ host language. If the data object is symmetric, the address in the pointer is called a *symmetric address*. All symmetric addresses are also local addresses and can, therefore, also be used for direct memory access.

Symmetric addresses can additionally be used for inter-process data transfers using OpenSHMEM routines with three restrictions.

```c
#include <stdlib.h>
#include <shmem.h>

// SDO: Symmetric data object
// PDO: Private data object

int global1; // SDO
static short global2; // SDO

int main (void) {
    char local1; // PDO
    static float local2; // SDO

    long* alloc1 = malloc(sizeof(long)); // PDO
    double* alloc2 = shmem_malloc(sizeof(double)); // SDO

    return 0;
}
```

Code 2.1.: Example declarations/allocations of symmetric and private data objects.

- While manipulation of the symmetric address passed to OpenSHMEM routines is valid, including pointer arithmetic, array indexing, and access of structure and union members, the addressed memory must entirely remain within a symmetric data object.
- Symmetric addresses passed to OpenSHMEM routines must be aligned according to their type or fixed size except for the "mem" interfaces, e.g., `shmem_putmem`.
- Symmetric addresses are not transferrable between PEs and are only valid in the PE they were generated on. Using a symmetric address on any PE other than the origin PE on which it was generated results in undefined behavior.

To summarize, the use of symmetric and local addresses should be mostly transparent to the application developer, and the management of symmetric data objects is similar to the memory management in C/C++. The application developer only needs to ensure that OpenSHMEM routines are only passed memory from within a symmetric data object when accessing another PE. However, the distinction between local and symmetric addresses is vital for OpenSHMEM's memory model and this thesis's data race detection approach.

2.2.2. Atomicity guarantees

OpenSHMEM also provides several routines that perform atomic operations on symmetric data objects. These operations guarantee that concurrent accesses by any atomic routines "to the same location, using the same datatype [...], and using

communication contexts [...] in the same atomicity domain will be exclusive." [8, Ch 3.2]

A *communication context* in OpenSHMEM defines an independent ordering and completion environment for operations, and each context is associated with a group of PEs. OpenSHMEM refers to these groups of PEs as *teams*, such as the predefined team of all PEs `SHMEM_TEAM_WORLD`. In other terms, contexts allow a user to specify independence or isolation of operations for communication within a team of PEs. This independence may be desirable in threaded environments or to overlap communication and computation more efficiently. While a context is an optional parameter that can be passed to non-collective OpenSHMEM routines, all non-collective operations are associated with a context. The operations which have not been explicitly passed a context operate on the default context `SHMEM_CTX_DEFAULT`. Further, contexts are considered to be in the same *atomicity domain* if their associated teams are all split by (possibly recursive) calls to a `shmem_team_split_*` routine from a common predefined team. Teams and contexts will be mostly disregarded for the remainder of this thesis, as the necessary tracking would result in a considerable increase in complexity for the later data race detection implementation in MUST.

However, the concept of teams and communication contexts is especially relevant to atomic OpenSHMEM operations as the exclusivity of the operations depends on a common atomicity domain. Any situation where the conditions mentioned at the start of this section are not met when using atomic OpenSHMEM operations will result in undefined behavior. The specification explicitly lists several scenarios where atomic OpenSHMEM operations are used and result in undefined behavior [8, Ch 3.2]:

1. Concurrent accesses to the same locations by atomic OpenSHMEM operations using contexts whose associated teams do not share a common predefined team.

2. Concurrent accesses to the same locations by atomic OpenSHMEM operations using different datatypes.

3. Concurrent accesses to the same locations by atomic OpenSHMEM operations and non-atomic OpenSHMEM operations.

4. Concurrent accesses to the same locations by atomic OpenSHMEM operations and non-OpenSHMEM operations (e.g. load/store operations such as `x+=5;`).

As teams and communication contexts increase the complexity of an OpenSHMEM application and require much tracking effort for an analysis application, they will be ignored for the data race detection approach and prototype in this thesis. Therefore all operations in our data race examples and models will happen in the default context within a single atomicity domain. However, consideration of their effects will be given where appropriate, and it should only affect the data race model marginally.

2.3. OpenSHMEM API

The OpenSHMEM API is designed similarly to MPI and other library specifications. The functionality is provided through function calls or macros defined in the `shmem.h` header with additional non-standard functionality provided in the `shmemx.h` header. All of these functions can easily be identified through the `shmem/shmemx` prefix and are split by the specification into ten groups according to their purpose [8, Ch 2]:

1. Library management
2. Symmetric data object management
3. Communication management
4. Team management
5. Remote memory access (RMA)
6. Atomic memory operations (AMOs)
7. Signaling operations
8. Synchronization and ordering
9. Collective communication
10. Mutual exclusion

For the data race detection in this thesis, the library, communication, and team management functions are mostly irrelevant. Therefore they will not be discussed here in detail and can be referenced in the official specification [8].

In general, OpenSHMEM is designed to emulate the usage patterns of MPI for everything other than communication. This includes library initialization, thread support, and runtime management like id lookup on PEs, among other things. A small example of an OpenSHMEM program can be referenced in Code 2.2.

2.3.1. Memory management

Memory management in OpenSHMEM may provide some pitfalls for a developer who is used to MPI. While private data objects are managed as expected in C, shared data objects can be created using two different methods. The implicit creation of shared data objects is described in Section 2.2 as these shared data objects exist for the entire runtime of the application as they are static or global variables.

The explicit creation can be done using several OpenSHMEM routines. These memory management routines are closely related to functions from the C standard library. The programmer can allocate, reallocate, align, and free shared data objects. They mirror the syntax and expected behavior of the C functions, except that the

```c
#include <shmem.h>
#include <stdio.h>

int main( int argc, char** argv ) {
    static int data;

    shmem_init();

    if( shmem_my_pe() == 0 )
        shmem_int_p( &data, 42, 1 );

    shmem_barrier_all();

    if( shmem_my_pe() == 1 )
        printf( "Integer received from PE 0 is %d\n", data );

    shmem_finalize(); return 0;
}
```

Code 2.2.: An example of an OpenSHMEM program. PE 0 transfers an interger to PE 1.

```c
void *shmem_malloc(size_t size);
void shmem_free(void *ptr);
void *shmem_realloc(void *ptr, size_t size);
void *shmem_align(size_t alignment, size_t size);
void *shmem_malloc_with_hints(size_t size, long hints);
void *shmem_calloc(size_t count, size_t size);
```

Code 2.3.: OpenSHMEM memory management routines.

return pointer is a symmetric address and they must be executed collectively. All of the memory management functions are listed in Code 2.3

Compared to other collectives described in later sections, these functions do not offer the option to specify a team of PEs. Therefore, all these functions must always be called by all PEs collectively. A better intuition may be that a programmer can replace all the C standard library functions with the OpenSHMEM functions for memory management without affecting their program behavior, as symmetric addresses can also be used for local memory operations. However, in OpenSHMEM, these data objects are allocated on the symmetric heap, the size of which is controlled through the environment variable `SHMEM_SYMMETRIC_SIZE` as mentioned in Section 2.2.

2.3.2. Data exchange routines

At the center of OpenSHMEM is one-sided communication as described in Section 2.1. For this, the OpenSHMEM API provides many routines which can be used to exchange data between PEs. These data exchange routines can be grouped by their

general functionality. There are *Remote Memory Access* (RMA) routines which are point-to-point routines to exchange contiguous or regularly stridden data. *Atomic Memory Operations* (AMOs) are point-to-point routines that can read, update, and write to a single element of a symmetric data object. Lastly, some collective routines exchange data within a team of PEs according to different strategies like broadcasts or reductions.

The standard blocking versions of all these operations are *locally blocking*. Locally blocking operations only return when all buffers can safely be (re)used. Therefore the return of a locally blocking routine can indicate two things. If the operation fills a local buffer, then the return of the routine indicates that the buffer has been filled correctly and data can safely be read from it. If the operation reads from a local buffer, the return of the routine indicates that the buffer can safely be reused. For example, in a locally blocking put, this could indicate that the data has been transferred or buffered by library runtime.

In order to correctly refer to all essential parameters of these routines, we need to introduce some terminology. Therefore we define the following terms for use in this thesis:

Origin The origin is the PE from which a point-to-point operation originates, or simply the PE calling the API function.

Target The target is the PE whose memory is accessed in the operation and which is specified in the API function.

Source The data object from which the operation reads.

Destination The data object which is written to by the operation.

Root The PE from whose data is distributed in a broadcast routine.

The origin and target are not mutually exclusive and can be the same PE when the origin issues an operation to itself.

RMA routines form the core of the OpenSHMEM API. With them, a developer can *put* data into or *get* data from a symmetric data object on a PE. This functionality is provided for all the basic numeric types specified in the C standard, an explicit list of which is given in the specification [8, Table 5]. RMA routines are also supported for fixed-size blocks of memory doubling in size from 8 to 128 bits and a generic *mem* interface. The general signature of RMA routines follows two predictable patterns. As mentioned in Section 2.3, all SHMEM routines are prefixed using `shmem_`, followed by the type name, and then either put or get.

Figure 2.4 shows the function signatures of the locally blocking versions of put and get operations for contiguous data. OpenSHMEM offers other versions of these operations as well. These are versions for single elements, stridden data, and non-blocking routines. While the first two are convenience features, non-blocking routines differ in their behavior. Usually, the buffers used by RMA operations are safe to use after the routines return and the data has been fetched. However, in non-blocking

```
void shmem_TYPENAME_put(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_putSIZE(void *dest, const void *source, size_t nelems, int pe);
void shmem_TYPENAME_get(TYPE *dest, const TYPE *source, size_t nelems, int pe);
void shmem_getSIZE(void *dest, const void *source, size_t nelems, int pe);
```

Code 2.4.: Example function signatures for RMA operations.

operations, designated with the `_nbi` suffix, this is not the case as the buffer can only be re-used following a call to `shmem_quiet`, which will be described in the following Subsection 2.3.3. All of the RMA operations can be referenced in the specification [8, Ch. 9.6].

Compared to RMA operations, Atomic Memory Operations (AMOs) may do more than just read or write data. They provide a communication mechanism that may combine read, modify, and write operations in an atomic manner if used correctly. The prerequisites for this are described in Subsection 2.2.2. However, in exchange for this atomicity of their operations, they only operate on single data elements. The specification provides a number of operations that should be familiar to anyone who has used atomic operations in other programming models before. AMOs are easily identified by the `_atomic` infix and similarly designed to RMA operations. They are again constructed using the `shmem_` prefix, followed by the datatype, and end with the operation type. For example, an atomic fetch&add operation or atomic compare&swap are shown in Figure 2.5. All AMOs can be referenced in the specification [8, Ch. 9.7].

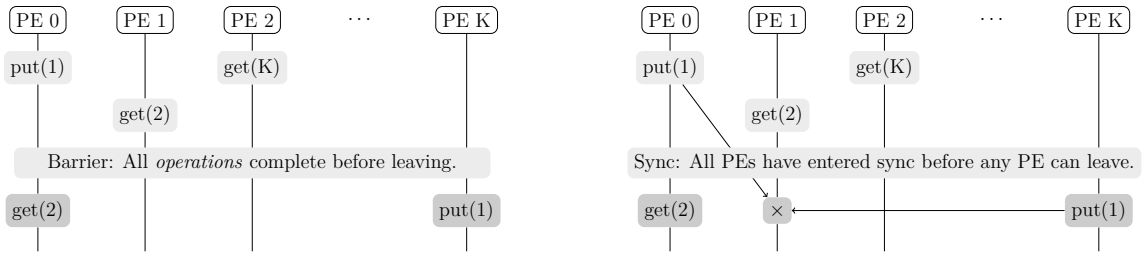
```
TYPE shmem_TYPENAME_atomic_fetch_add(TYPE *dest, TYPE value, int pe);
TYPE shmem_TYPENAME_atomic_compare_swap(TYPE *dest, TYPE cond, TYPE value, int pe);
```

Code 2.5.: Example function signatures for AMOs.

Lastly, OpenSHMEM provides a number of collective data exchange operations, these implement optimized communication strategies for common communication patterns, such as broadcasts or data reductions within a specific group of processes. They again follow the familiar pattern of `shmem_` prefix, datatype, and the operation's name. As the name implies, collective routines have to be executed by all PEs in the team specified in the routines. As they are collectively executed, these operations perform some process-level synchronization. This aspect will be further explored in Chapter 4 of this thesis. All collective operations can be referenced in the specification [8, Ch. 9.9].

2.3.3. Synchronization and memory consistency

The routines described in the previous section all access private and symmetric data objects in a one-sided nature. Therefore the interaction between PEs is reduced compared to a traditional message-passing model like MPI. This reduction in interaction may change the order of operations and the synchronization behavior of an



(a) Behavior of `shmem_barrier_all`.

(b) Behavior of `shmem_sync` routines.

Figure 2.2.: Behavior of collective synchronization routines.

application compared to an application using message passing. OpenSHMEM offers synchronization and memory consistency constructs closer to a typical shared-memory environment, such as barriers, fences, or locks.

For process synchronization, OpenSHMEM offers collective barriers. The most potent construct for synchronization among them and in OpenSHMEM as a whole is `shmem_barrier_all`. This barrier blocks until all PEs have called it and ensures the completion of local memory stores and remote memory updates issued through AMOs or RMA routines. While `shmem_sync` routines act similarly to the barrier, they *do not* enforce the completion of remote memory updates. In other terms, `shmem_sync` routines only perform process synchronization and do not enforce ordering across PEs. This difference between the barrier and sync construct is an important distinction and may lead to a race condition in an application. This situation is illustrated in Figure 2.2. Both subfigures show a collective synchronization routine. Subfigure 2.2a uses `shmem_barrier_all`, which also enforces completion of all local and remote operations. Subfigure 2.2b instead shows the same operations however using `shmem_sync_all`, as this does not enforce remote completion, the operations put(1) on PE 0 and PE K may create a race condition. The exact definition for `shmem_barrier` and `shmem_sync` routines can be referenced in the specification [8, Ch. 9.9].

While collective synchronization is helpful in many situations, there may be better strategies for a performance-conscious application, as it blocks all PEs from proceeding and introduces wait-states. In other situations, it may only be important that operations are delivered in a specified order or that operations complete remotely before a single PE can proceed.

For these cases, OpenSHMEM provides additional mechanisms to enforce such conditions. One of the simplest mechanisms for this is point-to-point synchronization routines, which can either wait or perform a non-blocking test for a condition on the origin PE. They work on the same types supported by standard AMOs as listed in the specification [8, Table 6] and can perform the standard numeric checks on these values as given in Table 2.1. These `shmem_wait_until` and `shmem_test` routines come in many variants, all of which are listed in the specification [8, Ch. 9.10]. They

Constant Name	Comparison
`SHMEM_CMP_EQ`	Equal
`SHMEM_CMP_NE`	Not equal
`SHMEM_CMP_GT`	Greater than
`SHMEM_CMP_GE`	Greater than or equal
`SHMEM_CMP_LT`	Less than
`SHMEM_CMP_LE`	Less than or equal

Table 2.1.: Point-to-Point Comparison Constants [8, Table 13]

are often combined with other synchronization mechanisms to enforce the order of some operations or indicate the completion of a remote operation on the local PE.

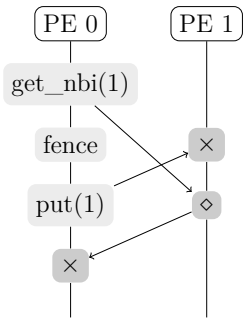
These mechanisms are `shmem_fence` and `shmem_quiet`, which are non-collective routines. Like memory fences in shared memory programming, `shmem_fence` prevents the reordering of OpenSHMEM operations. However, they behave differently. For example, `shmem_fence` creates a reordering fence for Put, AMO, store, and non-blocking Put routines issued to the same PE. In other terms, all writing operations with the same target issued before the fence on the origin will be delivered and visible before the operations issued after the fence. However, returning from the fence routine does not guarantee completion. Some possible orderings of operations with a fence are visualized in Figure 2.3.

In comparison, the `shmem_quiet` routine enforces stricter completion requirements. It is also a non-collective routine, but instead of only enforcing ordering for all writing operations on the tuple (origin, target), it forces the remote completion of all operations issued by the calling PE. Specifically, this includes *origin completion* and *remote completion*. Origin completion indicates that the accesses on the origin are completed and any subsequently issued operations can only see the effects of the previous operations. Similarly, remote completion indicates that all accesses on the target are compeleted and any subsequent operations will see the effects of the previous operations. Completion is also explicitly enforced for the non-blocking get routine, which the `shmem_fence` routine ignores. Therefore an ordering such as shown in Subfigure 2.3b where the put completes after the calling PE has returned from the `shmem_fence` routine is not possible if the fence routine is replaced with a quiet routine. This guarantees to the calling PE that all operations are visible to all other PEs before the calling PE can continue.

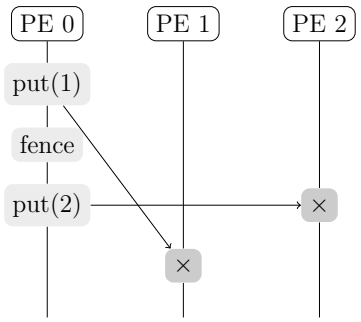
Lastly, OpenSHMEM also offers a locking mechanism on a symmetric data object of type `long` for exclusive access using the following routines defined in the specification [8, Ch. 9.12]:

```
void shmem_clear_lock(long *lock);
void shmem_set_lock(long *lock);
int  shmem_test_lock(long *lock);
```

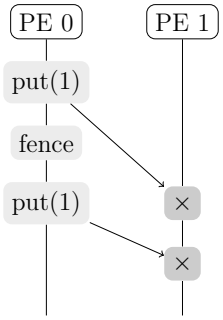
Code 2.6.: Example function signatures for AMOs.



(a) No order enforced between the non-block get and put on PE 0.



(b) No order enforced between both put operations and put to PE 1 completes after fence has returned.



(c) Fence enforcing completion order between two puts.

Figure 2.3.: Example completion orderings when a fence is used.

With these routines, typical mutual exclusion constructs between PEs are achievable, which behave as expected from a shared memory environment. Such mechanisms are already well studied. Therefore, this lock feature is not the focus of this thesis but will be included in the data race detection model.

3. Data race classification

In this chapter, this thesis will provide a classification of data races in OpenSHMEM. Data races are part of an error class typical in shared-memory parallel programs and programs containing other forms of concurrency. These forms of parallelism can range from explicit parallelism using threads, processes, or similar concepts common in HPC applications, to asynchronous operations, which are commonplace in a typical server application. Errors that result from this can manifest as unintended non-determinism or non-atomic execution of critical sections.

However, clearly classifying such erroneous behavior and providing a general classification for it has proven challenging as it depends on the specific concurrency and memory model used by the programming model [34]. Therefore before discussing the data race detection approach and implementation, we must first define what data races are in the OpenSHMEM programming model.

3.1. Race conditions and data races

In general, such errors have a long history in programming, as many applications contain some form of concurrency to improve performance. Performance can imply many measures, such as throughput or latency, which are essential in various applications. However, the terminology often differed depending on minor differences in the semantics of definitions and even just the domain in which such errors were examined. Terms such as *access anomaly*, *race condition*, *data race*, *critical race*, or *harmful shared-memory access* often refer to the same or very similar failures [34].

This problem was also identified in a paper by Netzer and Miller [34]. Their paper examined races considered failures in shared-memory parallel programs and aimed to unify terminology within the domain of shared-memory parallel programming and offer a formal model for such races. They characterize two types of races with the following:

- "General races cause non-deterministic execution and are failures in programs intended to be deterministic." [34]

- "Data races cause non-atomic execution of critical sections and are failures in (non-deterministic) programs that access and update shared data in critical sections." [34]

  Critical sections are blocks of code intended to be executed as if they were atomic. *Atomic* means that the final state of the variables accessed in the

section only depends on the initial state when entering the section and the operations performed within the section.

Netzer and Miller further introduce the concepts of *feasible* and *apparent* races, which apply to both kinds of races. The exact definition depends on the formal model, which can be referenced in the paper [34]. Intuitively they can be described as the set of all races which are desirable to detect and an easier-to-detect subset. Feasible races represent the set of all races that could occur based on the possible behavior of the program. This implies examining all possible interleavings of operations, which quickly explodes in complexity. In general, the problem of race detection is NP-hard [34].

Apparent races are a subset of the feasible races that can be used to approximate the set of feasible races. Instead of reasoning about all possible executions or orderings of operations, the definition of apparent races only uses explicit synchronization mechanisms such as the fork/join model or mutual exclusion. Therefore detecting apparent races is more manageable but not necessarily efficient. Detecting apparent races in programs implementing some form of mutual exclusion remains NP-hard [34].

The notion of general races is quite broad and only applies to programs intended to have deterministic execution. However, in a PGAS programming model like OpenSHMEM, where non-deterministic execution might be desirable, e.g., for performance optimizations, general races are not universally considered a failure. For this reason, we will not study general races in this thesis.

In contrast, the definition of data races by Netzer and Miller fits nicely into the kind of errors we wish to detect in this thesis. They are primarily the result of a programming error rather than a fundamental problem in the algorithm and are a "local" property of an execution [34]. Debugging such errors can still prove problematic as the visible manifestation of the data race to the developer may only show as an indirect failure, and the non-deterministic nature hampers the backtracing of the error to the data race. Therefore this thesis focuses on detecting *apparent data races*. From now on, we will refer to apparent data races simply as data races since an online detection mechanism does not lend itself to detecting all feasible races.

3.1.1. Data races in other programming models

Data races can occur in many programming models. However, they are especially prevalent in models that reflect shared-memory models. Examples of this are explicit threading in C/C++ [48, 49], shared-memory programming APIs like OpenMP [25, 24] or models which represent remote memory accesses such as MPI RMA [2, 45].

C programming language

Data races in the C programming language are particularly interesting to this thesis as they are closely related to OpenSHMEM. In the previous chapter, Section 2.2

describes the memory model used within OpenSHMEM. One of the most significant points was that *private data objects follow the C memory model*. This dependence on the shared memory model from C has notable implications for the definition of data races in this thesis as we need to consider existing definitions of data races for the C programming language [7, 18]. As all symmetric data objects are private data objects simultaneously, the definition for data races in this thesis must also cover the extensions placed upon the memory model by symmetric data objects. At the time of writing, the current standard for the C programming language ISO/IEC 9899:2018, from here on referred to as C17, defines a data race as follows:

- "The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior." [18, Ch. 5.1.2.4, 35]

In other words, an execution of a C program contains a data race if a memory location is accessed by two threads concurrently where at least one access modifies the location and at least one access is non-atomic. Code 3.1 is an example of a program that could exhibit such a data race. However, under this definition, only

```c
#include <threads.h>
#define NUM_THRDS 2

int thrdFunc(void* race_var) {
    *((int*) race_var) += 1;
    return 0;
}

int main(int argc, char* argv[]) {
    thrd_t thrdId[NUM_THRDS];
    int race_var = 0;

    for (int i=0; i < NUM_THRDS; ++i)
        if (thrd_create(&thrdId[i], thrdFunc, &race_var) !=
   thrd_success)
            return -1;

    for (int i=0; i < NUM_THRDS; ++i)
        thrd_join(thrdId[i], NULL);

    return race_var;
}
```

Code 3.1.: A data race on `race_var` exclusively using features from the C standard library.

data races that occur during an individual execution of a program are clearly defined. Therefore, all possible executions must be examined to define all possible data races similar to the set of feasible races.

This is very similar to the definition of data races provided by Netzer and Miller. As the two definitions from Netzer and Miller and the C17 standard are comparable, we can adapt the formal model provided in their paper [34] to the C memory model as required by OpenSHMEM.

OpenMP

Manual threading is common in many applications, such as server and client programs, where each thread has a different task. This represents unnecessary complexity for applications in HPC. Most of the time, within threaded HPC applications, all threads spread a single task by splitting up the data set and performing the same computations on different data. This pattern led to the development of OpenMP [9], which is a compiler directive-based API for programming shared-memory programs.

The core of OpenMP is formed by the `parallel` directive, which indicates that the following section or, more precisely, structured block of code is supposed to be executed by all threads of the OpenMP runtime in parallel. The programmer can then use additional directives to specify how data is passed to a thread. These specifications include whether the data is shared or private and which part of the data the thread is supposed to work on. All of this can also be achieved by using manual threading. However, OpenMP has proven to be the preferred way for many to program shared-memory parallel programs in HPC. OpenMP significantly reduces code complexity and improves the application developers' productivity as much auxiliary parallelization work is taken care of by the OpenMP runtime.

However, data races still occur and have been studied extensively in OpenMP. The community surrounding OpenMP has developed many tools [4] to aid developers in detecting data races, and benchmark suites [24] to test these tools have been developed. As the semantics of OpenSHMEM are closely related to shared-memory programming, the work done on data races provides a solid knowledge base that data race detection approaches in OpenSHMEM can study. For example, the Code 3.2 is a microbenchmark from DataRaceBench [24], a comprehensive data race test suite. This thesis uses DataRaceBench as an example to develop our test suite.

MPI RMA

Another programming model which proves helpful in the development of data race detection approaches in OpenSHMEM is MPI RMA. MPI traditionally focused on explicit message passing. However, one-sided communication was introduced in version 2.0 of the MPI standard [31]. One-sided communication in MPI is very similar to OpenSHMEM as they implement RMA as a communication mechanism. OpenSHMEM focuses on simplicity, only providing a small number of communication and synchronization routines, while MPI RMA is quite complex and fine-grained in controlling all operations. For example, OpenSHMEM has no mechanism to complete a single issued operation individually, and all PEs always have the same symmetric

```c
/*
A loop with loop-carried anti-dependence.
Data race pair: a[i+1]@64:10:R vs. a[i]@64:5:W
*/
#include <stdio.h>
int main(int argc, char* argv[])
{
  int i;
  int len = 1000;

  int a[1000];

  for (i=0; i<len; i++)
    a[i]= i;

#pragma omp parallel for
  for (i=0;i< len -1 ;i++)
    a[i]=a[i+1]+1;

  printf ("a[500]=%d\n", a[500] );
  return 0;
}
```

Code 3.2.: `DRB001-antidep1-orig-yes.c` from DataRaceBench 1.4.0 [23].

data objects since all allocations are collectives over the world team. In comparison, MPI RMA provides features with which this can be achieved.

The relationship between both programming models is even more profound than one-sided communication, as OpenSHMEM can be viewed as an abstraction of MPI RMA. This is especially apparent since there is work on implementing OpenSHMEM only using MPI one-sided communication [13]. For this reason, we rely on previous work in data race detection for MPI RMA [45] to develop our data race detection for OpenSHMEM.

This close relationship also implies that data races in both models are very similar. The previous work on detecting data race in MPI RMA using MUST, on which we base our approach, provides the example in Figure 3.1 of a data race in MPI RMA. Based on this, we use both the semi-formal data race model and the implementation within MUST as a starting point for data race detection in OpenSHMEM.

3.1.2. Data races in OpenSHMEM

In the previously discussed programming models, the mentioned errors fall under the characterization of data races, as given by Netzer and Miller. In every case, the communities surrounding these models have also given definitions of data races which are once again similar but with special consideration given to their respective models. However, at the time of writing and to the best of our knowledge, previous works on OpenSHMEM correctness [14, 37, 5] have not provided any definition for data

P0 (origin)	P1 (target)
	win location X
Barrier	Barrier
Win_lock(P1)	
...	
buf=42	...
Put(&buf,P1,X)	X=1
...	...
Win_unlock(P1)	
Barrier	Barrier

Figure 3.1.: Data race across processes in MPI RMA: P0 performs a remote write to location `X` using an `MPI_Put` call that is concurrent to the `X=1` instruction at P1. The value of the variable `X` is undefined due to a data race. [45, Fig. 1]

races within OpenSHMEM. To this end, we define a data race in an OpenSHMEM program as follows:

Data race: An execution of an OpenSHMEM program contains a *data race* when two or more entities perform *unsynchronized conflicting accesses* to the same data object.

Entities are threads or PEs acting through the library. *Conflicting* means that the accesses have overlapping memory locations in the data object and at least one of the accesses is a write/store access. *Unsynchronized* means that the accesses are concurrent and not protected by an appropriate synchronization mechanism. Missing synchronization in OpenSHMEM can occur in many situations. Some examples include missing mutual exclusion mechanisms such as locks, unaligned access for types in OpenSHMEM routines, or mixed atomic OpenSHMEM routines.

This definition is expanded upon in the following Chapter 4 using a semi-formal adapted from the existing work on MPI RMA data race detection in MUST [44, 45]. However, using this definition, we can examine the semantics in the OpenSHMEM specification and infer situations in which data races can occur. For example, in the Code 3.3, we can apply the data race definition as follows. The two entities are PE 0 and PE 1, accessing the symmetric data object `data` on PE 2. This access is conflicting as one operation is a put operation. Those operations have no barrier, fence, or similar synchronization mechanism separating them. Thus, the accesses are unsynchronized. Therefore, this code contains a data race.

3.2. OpenSHMEM data race classification

There are many more situations in which data races within an OpenSHMEM program can occur. Among these are incorrect usage of atomic operations as described in

```
16      if( my_pe == 0 )
17          shmem_int_p( &data, 1, 2 ); // CONFLICT
18
19      if( my_pe == 1 ) {
20          int fetched = shmem_int_g( &data, 2 ); // CONFLICT
21          printf( "Data on rank 1 is %d\n", fetched );
22      }
```

Code 3.3.: A data race with conflicting put and get operations. Test case put-get-remote from this thesis's data race set in Section 3.3.

Section 2.2.2 or use of the `shmem_fence` routine, which does not enforce completion, instead of the `shmem_quiet` routine, to name a few. For example, the Code 3.3 is a straightforward data race using the Remote Memory Access routines from the OpenSHMEM API. As explained before, this data race occurs since there is no synchronization mechanism to enforce an ordering of both operations. However, when using Atomic Memory Operations, this concurrency may even be desired, and the specification of these operations affords an exclusivity guarantee. Therefore, replacing the RMA operations in Code 3.3 with the respective `fetch` and `set` AMOs will result in a program that does not contain a data race. This change is minor to a developer who merely wants to achieve the data movement without introducing an error. Nevertheless, from the viewpoint of correctness tools for OpenSHMEM, these differences are significant as entirely different synchronization mechanisms are used, and the tool must be able to understand all the synchronization mechanisms to analyze the program correctly.

For this reason, test cases or test suites are created to benchmark performance and verify the functionality of correctness tools. In particular, some test suites include data race test cases or are entirely designed for data races in other programming models like MPI or OpenMP. A well-known test suite is the *MPI Bugs Initiative* [22], which, among other tests, contains data race tests for MPI RMA. It has been used in previous work on data race detection in MPI RMA [45] to evaluate the classification quality of the developed tool. Similarly, DataRaceBench is an open-source benchmark suite designed to systematically and quantitatively evaluate the effectiveness of data race detection tools [24]. It is focused on programs written in OpenMP and as of DataRaceBench version 1.4.0 [23] contains 344 tests. DataRaceBench is used to evaluate the classification quality of shared-memory data race detectors such as ThreadSanitizer [48] and compare their effectiveness.

For OpenSHMEM, no such test suite, which includes comprehensive coverage of data races, is provided by previous work [14, 5, 38]. Thus, we created a set of small programs presented in the next Section 3.3 for this thesis. The focus of this set is to cover as many different situations as possible to test our implementation. During the development of this set, we observed that many test cases share properties and patterns. The authors of DataRaceBench made a similar finding and used a labeling system to classify their microbenchmarks [24, Table 2]. They used two sets of labels,

one for the microbenchmarks containing a data race and one without data races. These labels indicate the different properties of a microbenchmark, such as the use of a specific feature of OpenMP.

For our data race set, we apply a similar system to classify our test cases. However, instead of using separate labels based on the presence of a race, we separate the labels into *race properties* and the involved *synchronization mechanisms*. Labels in the *race property* group describe the data race itself. For example, the label 'Safe' indicates that no data race is present in the test case, while the label 'Local buffer race' indicates a data race on the origin of an OpenSHMEM routine. The labels in the group *synchronization mechanisms* indicate the usage of an OpenSHMEM synchronization mechanism. An example is the label 'Atomics', which indicates that in the example code, OpenSHMEM AMOs are used. Each label is given a description below:

Race property labels

Safe: The test case contains no data race.

Local buffer race: The data race is on the *origin* PE that results when a communication operation has not been locally completed, and the origin process performs a conflicting access. This can be a write access to the local source buffer or any access to the local destination buffer that results in undefined behavior. Code 3.4 gives an example of a local buffer race. Local buffer races were adapted from the MPI RMA data race detection in MUST [45].

```
1 shmem_int_put_nbi(dest, &source, ...);
2 source += 1;
```

Code 3.4.: A local buffer race between a put operation and a local store.

Remote race: The data race is on the *target* PE that results when a communication operation has not been remotely completed and another entity performs a conflicting access. This can be a write access to the remote source buffer or any access to the remote destination buffer that results in undefined behavior. Code 3.5 gives an example of a remote race. These two processes remotely write to the same memory on a third remote process. Remote races were adapted from the MPI RMA data race detection in MUST [45].

Unreliable manifestation: The test case contains a data race, but the concurrency of the conflicting operations depends on an external factor, e.g., the order in which a lock is acquired. This situation is visualized in Figure 3.2a. In this example, if PE 1 acquires the lock first, then the put operation it has issued must be completed due to the `shmem_quiet` call, and the PEs perform process synchronization through the lock acquisition. Thus, the put operation on PE 1

```
if( mype == 0 ) {
    shmem_int_put_nbi( dest, &source0, 1, 2 );
} else if( mype == 1 ) {
    shmem_int_put_nbi( dest, &source1, 1, 2 );
}
```

Code 3.5.: Remote race of two put operations writing to the same destination.

must be completed before the put operation on PE 0 is issued, and therefore no data race is present in this schedule. However, if PE 0 acquires the lock first, both operations can be issued concurrently, resulting in a data race.

False atomicity: The test case uses atomic operations, but the OpenSHMEM specification does not grant exclusivity. The possible causes are described in Section 2.2.2, and an example is shown in Figure 3.2b where the operations are not atomic as they use different datatypes.

Inconspicuous reordering: The test case uses some synchronization mechanism in the program, but operations can be reordered across it. For example, this can be the case for memory ordering routines as illustrated in Figure 3.2c.

PE 0	PE 1
`set_lock(&lock);`	put(2); quiet();
...	set_lock(&lock);
clear_lock(&lock);	...
put(2); quiet();	clear_lock(&lock);

(a) An example where the manifestation of a data race depends on the order of the lock acquisition.

PE 0	PE 1
`int_atomic_set(2);`	long_atomic_set(2);

(b) Conflicting OpenSHMEM atomic operations using different data types.

PE 0	PE 1
`put(2);`	
fence();	fence();
	put(2);

(c) Conflicting put operations across fence routines.

Figure 3.2.: Illustrations of race property labels.

Synchronization mechanisms labels

Process-based synchronization: The test case uses collective routines to synchronize PEs. These collectives are the barrier, team sync, and other collective routines.

Memory ordering routines: The test case uses memory-ordering routines to order or complete communication operations. These routines are fence and quiet.

Resource-based synchronization: The test case uses resource-based synchronization mechanisms. This includes the lock routines, signaling operations, and point-to-point synchronization routines.

Atomics: Atomic operations are used within the test case.

Each test case is associated with at least one race property label. However, more labels can apply. The vision behind this labeling approach is that the data race detection implemented in MUST and, potentially, in the future other data race detection tools for OpenSHMEM can be evaluated on the data race set and that the labels can provide a deeper insight into their classification quality. This labeling approach can be used to be more fine-grained with these measurements and, for example, be able to tell in which situations data race tools are better at identifying races. For example, if one tool has fewer false alerts on remote races than another.

3.3. Data race set

As mentioned in the previous Section 3.2 in previous work on OpenSHMEM correctness, data races have been considered an error class. However, the other correctness tools OpenSHMEM-Analyzer [14] and OpenSHMEM-Checker [5], have only provided a few tests containing data races, as both tools are static analyzers and do not focus on data race detection. Therefore, we developed a set of data race test cases in OpenSHMEM. This test set will be used in the last Chapter 6 to evaluate the performance of our data race detection implementation for OpenSHMEM. A table A.1 of all test cases is given in the appendix with their respective labels.

All the test cases conform to some ground rules in order for the tests to remain relevant to OpenSHMEM data races. Each test case is a single file compiling OpenSHMEM program and may contain a data race. We restrict the data races to races that involve OpenSHMEM functionality. To the best of our knowledge, no test case contains any other error. To further restrict any external factors, each test case has the following design pattern:

The `shmem_barrier_all` routine is the strongest synchronization mechanism within OpenSHMEM, as it performs both process synchronization and completes all outstanding RMA operations and memory stores. Therefore the code outside these barriers does not affect the test code between them. Our OpenSHMEM correctness checking implementation described in Chapter 5, however, is unaware that these barriers serve such a purpose, and they do not get any special handling.

Furthermore, each program file for a test case is named according to the pattern `<race-name>_<race|safe>.c`. The `<race-name>` roughly describes the test case. For example, the test case `atomic-mixed` uses both atomic and non-atomic operations. Lastly, `<race|safe>` indicates whether a program file contains a race.

The entire test suite is available in the artifacts of thesis [19].

```
// #include <> directives

int main(int argc, char* argv[]) {
    // Test setup

    shmem_barrier_all(); // Test start

    // Test code

    shmem_barrier_all(); // Test end

    // Post-processing
}
```

Code 3.6.: Design pattern of all test cases.

4. OpenSHMEM data race detection model

The previous Chapter 3 introduced data races in OpenSHMEM and other programming models. In many of these models, the prose data race definition of 'unsynchronized conflicting memory accesses at least one which is modifying' is easily applicable. However, the semantics of these programming models often result in situations where defined behavior in one model is undefined behavior in another. For example, in OpenSHMEM, all publicly accessible memory is also accessible by operations not issued through OpenSHMEM routines like a local store. In comparison, MPI RMA differentiates between *public* and *private* window copies, which are potentially updated differently depending on the memory model.

To better reason about the specifics of data races and their detection, this chapter introduces a semi-formal model for MPI RMA and, based on the similarities between OpenSHMEM and MPI RMA, adapts it to OpenSHMEM. Additionally, based on a data race detection model developed using the semi-formal model for MPI RMA, this chapter also introduces an on-the-fly race detection model for OpenSHMEM.

4.1. Semi-formal model for MPI RMA

This section introduces the semi-formal model of the MPI RMA semantics proposed by Hoefler et al. [17]. Their paper proposes a formalization of the memory and synchronization semantics described in the MPI-3 standard document [32]. This work is intended to maintain programmers' readability while remaining expressive and precise enough to allow researchers to further derive formal specifications and valid transformations for programs using MPI RMA. However, the model by Hoefler et al. [17] does make some admissions. As stated in the paper, the main reason for these admissions is to simplify the notation to achieve the model's human-readability goal.

As MPI RMA and OpenSHMEM are semantically very similar [13], this model can be adapted and applied to OpenSHMEM. Fortunately, the limitations in the notation are irrelevant to the semantics of OpenSHMEM as there is no notion of active-target synchronization or request-based communication operations within OpenSHMEM.

4.1.1. Model description

A characterization of the model is needed to apply the semantics of OpenSHMEM to the model by Hoefler et al. [17]. However, fully reproducing the model here would be redundant as it can be referenced in the paper [17] and in previous work on data race detection in MPI RMA [44, 45].

The model describes a *correct* MPI RMA program as "a program where each conflicting access is synchronized with process synchronization (we call this *happens* before) as well as memory synchronization (we call this *consistency*)." [17] These synchronizations are achievable using a variety of calls in MPI RMA, and similar capabilities exist within OpenSHMEM. However, correctly using these synchronization mechanisms does not necessarily result in deterministic program execution. Therefore the model considers the execution of programs.

Actions

The fundamental unit of this model is formed by *actions*. In general, actions model the individual operations executed and effects experienced by the processes in RMA communication and synchronization calls, as well as local memory operations such as loads and stores. The model differentiates between two different kinds of actions *memory acctions*, and *synchronization actions* following previous definitions in work on the Java and C++ memory models [7, 30]. *Memory actions* are defined as the tuples

$$\langle a, o, t, rl, wl, u, p \rangle$$

with the following elements [17]:

- a: [action type] can be one of the following: local store (*ls*), local load (*ll*), remote communication put (*rcp*), remote communication get (*rcg*), remote accumulate get (*rag*, with the special case fetch and op), remote accumulate (*rac*), or remote accumulate compare and swap (*ras*).
- o: [origin] contains the MPI process rank for the origin of the action.
- t: [target] contains the MPI process rank for the destination of the action. Actions of type *ll* and *ls* can have only the local process as destination.
- rl: [read location] contains the location read by the action. This is not specified for ls and is a tuple of the form $\langle$compare location, swap location$\rangle$ for *ras*.
- wl: [write location] contains the location written by the action (not specified for *ll*).
- u: an arbitrary [unique] identifier for the action.
- p: [source point] is a label identifying the source program point.

Similarly, a *synchronization action* is defined as the tuple

$$\langle a, o, t, u, p \rangle$$

with the following elements [17]:

a: [action type] can be one of the following: fence (*sfe*), lock shared (*sls*), lock exclusive (*sle*), unlock (*sul*), lock all (*sla*), unlock all (*sula*), flush (*sfl*), flush local (*sfll*), flush all (*sfla*), flush local all (*sflla*), win-sync (*sws*), and external synchronization (*ses*, e.g., matching send/recv pairs or collective operations).

o: [origin] contains the MPI process rank for the origin of the action.

t: [target] contains the MPI process rank for the destination of the action. The actions *sla*, *sula*, *sfla*, *sflla* have a special identifier $\star$ as destination, which stands for the entire set of processes associated with the window.

u: an arbitrary [unique] identifier for the action.

p: [source point] is a label identifying the source program point.

In short, each memory action is uniquely defined through a tuple that tracks the type of the memory action, which memory locations are accessed, and from where in the code this action originates. Similarly, synchronization actions are uniquely defined through a tuple that tracks the type of synchronization, the origin, the target(s), and the source location of the synchronization action.

For brevity, these actions are generally named according to their types. Therefore, an action x where $x.a = z$ is referred to as z, for example, *ls* would read "the action *ls* of type *local store*". Additionally, the paper also introduced some shorthands for actions of similar types. They are akin to some notations for regular expression, where $v = (x|y)$ indicates that v is either of type x or y and $*$ is used as a wildcard. For example, $rc*$ would indicate that the action is a *remote communication* action and is defined with $rc* = (rcp|rcg)$. Additional abbreviations for common types are defined within the paper; however, their use is straightforward. For example, $s*$ identifies a synchronization action, as synchronization actions exclusively use the s prefix.

With these actions, the explicit statements in a program execution, such as a put operation, can be modeled. Nevertheless, some interactions cannot be modeled only using these actions alone. By their nature, RMA operations are one-sided, and a target process does not need to interact explicitly in the operation when using passive-target synchronization. To model the effects at the target of remote memory actions $r*$, unlock actions $(sul|sula)$, and flush actions $(sfl|sfla)$, *virtual actions* $(vac|vas)$ are created at the target process. A virtual communication action *vac* represents the event when the effect of the remote memory action takes place at the target. For example, for a remote communication put action or *rcp*, the corresponding virtual communication action *vac* commits the data to the destination memory. Similarly, a virtual synchronization action *vas* represents the event when memory synchronization occurs at the target. These virtual actions are necessary to define the consistency and process order for purely one-sided remote events.

Executions of MPI RMA programs

Using these actions, Hoefler et al. [17] define an execution of a program as a set of actions, orders, and functions

$$X = \langle P, A, \xrightarrow{po}, [...], \xrightarrow{so}, \xrightarrow{hb}, \xrightarrow{co} \rangle$$

where

- P: is the program to be executed;
- A: is the set of all actions (types $s* | r* | l* | va*$);
- $\xrightarrow{po}$ specifies a total order (program order) of actions ($s* | r* | l*$) at the same process much like the "sequenced-before" order in C++ or the "program order" in Java, which specifies the order of executions in a single-threaded execution; [...]
- $\xrightarrow{so}$ is a partial order of the synchronization relations of process synchronization and virtual actions ($s* | va*$) including external waiting-for relationships (e.g., arise from *ses* actions like collective operations and matched send/recv pairs);
- $\xrightarrow{hb}$ is a transitive relation between pairs of actions, in which the relation $\xrightarrow{hb}$ is the transitive closure of the union of $\xrightarrow{po}$ and $\xrightarrow{so}$; and
- $\xrightarrow{co}$ is a partial order of memory actions and virtual actions ($r* | l* | va*$), in which a consistency edge $x \xrightarrow{co} y$ guarantees that the memory effects of action x are visible to y.

Within this definition, the consistency order $\xrightarrow{co}$ and happens-before $\xrightarrow{hb}$ are particularly interesting. They represent the concepts of memory consistency and process synchronization, respectively. Both memory consistency and process synchronization are required to ensure correct executions. However, using passive-target synchronization in MPI RMA and, similarly, in OpenSHMEM, synchronization mechanisms control these orders individually. Therefore, to abbreviate, when two actions x and y are ordered in both the consistency and happens-before order, the combined *consistent happens-before order* is defined as [17]:

$$x \xrightarrow{cohb} y := x \xrightarrow{hb} y \wedge x \xrightarrow{co} y$$

Lastly, for any orders $R \in \{\xrightarrow{po}, \xrightarrow{so}, \xrightarrow{hb}, \xrightarrow{co}\}$ and two actions x and y we denote that they are not ordered by R using:

$$x \parallel_R y := \neg(x \xrightarrow{R} y \vee y \xrightarrow{R} y)$$

For example, $x \parallel_{hb} y$ means that actions x and y happen concurrently in the happens-before order and respectively $x \nparallel_{hb} y$ means that the actions are not concurrent. Such short notations and abbreviations are also used for the other orders within an execution.

Consistency and synchronization of virtual actions

In an execution, the actions are not arbitrarily ordered. Instead, actions are ordered according to the specifications in the model. These connections between actions are rigorously specified by Hoefler et al. [17] for the functionality of MPI RMA. However, a general understanding, especially for concepts relevant to OpenSHMEM, can be gained using more straightforward rules which describe a subset of the specifications and the semantics of OpenSHMEM as described in its specification document [8].

Virtual actions are one of these concepts central to understanding the model. As previously mentioned, they are the counterpart of RMA communication and synchronization action on the target process and model the effects of those operations on the target. Therefore, each virtual action is linked to the action on the origin from which it originates. Specifically, regardless of whether it is a virtual communication or synchronization action, each virtual action is both *consistent* and *process synchronized* with its matching action at the origin. This is required to correctly reason about the happens-before and consistency order on the target with their transitive property.

This is modeled by splitting communication actions $r*$ and synchronization actions $s*$ into a *start* action $r*_s$ or $s*_s$ and *end* actions $r*_e$ or $s*_e$ and relate them to the matching virtual action as follows:

$$r*_s \xrightarrow{coso} vac \quad \text{and} \quad vac \xrightarrow{coso} r*_e$$
$$s*_s \xrightarrow{coso} vas \quad \text{and} \quad vas \xrightarrow{coso} s*_e$$

Which is abbreviated by Hoefler et al. into the rules

$$r* \xleftrightarrow{coso} vac \tag{R1}$$
$$s* \xleftrightarrow{coso} vas \tag{R2}$$

using the short notation. A virtual action $va*$ happens at the target $(s*|r*).\mathrm{t}$ of its originating action $(s*|r*)$. The location where a virtual action happens is its origin $va*.o$.

4.1.2. Data races in MPI RMA

Using virtual actions, both local and remote effects of remote communication in a program execution can be modeled. With this, the necessary information about an execution can be encoded in the model, and Hoefler et al. [17] use it to define *valid executions* of a program using MPI RMA. They do this by specifying the orders between the actions for an execution of a program to be valid, which is quite complex to do rigorously. However, within this thesis, we are only interested in data races and their detection.

Fortunately, Hoefler et al. [17] define data races. However, this definition depends on the memory model used within MPI for RMA operations. MPI supports two different memory models, the *separate memory model* and the *unified memory model*. In the separate model, RMA operations and local operations operate on a public and

private copy of the same variable instance and may require additional synchronization to propagate updates to these copies. In comparison, in the unified memory model, public and private copies are identical, and update through RMA communication calls or local updates are observed *eventually* without additional RMA calls. However, OpenSHMEM is similar to the unified memory model, as all symmetric data objects are private data objects. Therefore we only consider the definition for data races in the unified memory model.

In the unified memory model, two memory actions x and y are called *conflicting* if they are directed toward overlapping memory locations at the same process, and either one of the two operations is a put, precisely one of the operations is an accumulate, or one operation is a get and the second one is a local store [17]. This forms the precondition for the possibility of a data race. Ultimately, Hoefler et al. [17] define a data race as follows: A data race between two conflicting operations x and y exists if they are not ordered by both $\xrightarrow{hb}$ and $\xrightarrow{co}$ relations

$$\neg(x \xrightarrow{cohb} y \vee y \xrightarrow{cohb} x)$$

that is,

$$x \parallel_{hb} y \vee x \parallel_{co} y.$$

In other words, a program is data race free if all conflicting accesses are ordered by $\xrightarrow{cohb}$. This maps well onto the data race definition for OpenSHMEM in Section 3.1.2.

4.2. OpenSHMEM semantics

In order to apply this model to OpenSHMEM and use it in this thesis to perform data race detection, we need to identify the similarities and differences in the semantics of both programming models. For this, we make a few general assumptions to reduce the MPI RMA model to what is necessary to model OpenSHMEM. As mentioned previously, the concepts of active-target synchronization or request-based communication operations do not exist within OpenSHMEM and are, therefore, not considered in our OpenSHMEM model. However, In OpenSHMEM, all symmetric data objects on which OpenSHMEM routines operate are private data objects simultaneously. Thus, OpenSHMEM does not separate accesses issued through OpenSHMEM routines and local memory accesses made by other parts of the program. This fits the definition of the unified memory model available within MPI RMA. Therefore, for the remainder of this chapter, MPI RMA is discussed in the context of a unified memory model and using passive target synchronization.

However, there is one exception to this pattern. Although OpenSHMEM does not explicitly use active-target synchronization, the special case of `shmem_barrier_all` exists in OpenSHMEM 1.5 [8, Ch. 9.9.1]. This barrier performs process synchronization and enforces the completion of operations. Thus, it resembles the functionality of an `MPI_Win_fence`, but the barrier in OpenSHMEM is restricted to only operate on all PEs collectively and splits the execution of an OpenSHMEM program into two

independent regions. For this reason, we will handle `shmem_barrier_all` as a special case in Section 4.2.2. Lastly, OpenSHMEM exclusively uses one-sided communication and provides no functionality besides collective synchronization and communication routines. Therefore, we will not explicitly model external synchronization actions *ses* for OpenSHMEM. However, the process synchronization order $\xrightarrow{so}$ for such cases is given in the MPI RMA model [17].

Under these assumptions, we define actions for OpenSHMEM routines as was done for MPI RMA. The actions are defined using the same tuple, using different action types to differentiate actions that reflect OpenSHMEM routines from the previously defined action types. For memory actions, we use the following types in our model for OpenSHMEM:

- Local memory accesses are represented using the action types local load (*ll*) and local store (*ls*) as done by Hoefler et al.
- OpenSHMEMs remote communication routines are represented using the types remote communication put (*rcp*) and remote communication get (*rcg*). However, OpenSHMEM also provides non-blocking versions of these operations, and therefore we defined the types of the non-blocking versions as (*rcpn*) and (*rcgn*) for non-blocking put and get operations, respectively.
- OpenSHMEMs atomic operations are represented very simply, using load and store actions, as the other operations can be composed of them. For data race detection, this composition of concurrent atomic operations does not influence the result as they will not cause a data race by themselves, and if one conflicting action results in a data race, then the total operation would result in a data race. Therefore we use the types remote atomic load and store (*ral*, *ras*) and their non-blocking versions (*raln*, *rasn*).

Similarly, the synchronization mechanisms provided by OpenSHMEM are different from the synchronization calls provided in MPI. Therefore, we define the following synchronization action types:

- We define the barrier action (*sb*) for `shmem_barrier_all`, which performs processes synchronization like the MPI barrier but also enforces origin and target completion of all operations.
- In comparison, we define the sync action (*ss*) for sync routines like `shmem_sync` in OpenSHMEM, which perform process synchronization, but do not enforce any completion.
- Additionally, the memory consistency routines offered by OpenSHMEM are represented using the fence action (*sf*) and the quiet action (*sq*).

Using these action types, we can now define how these actions and virtual actions are ordered in an execution. We do this explicitly for the most prominent communication

and synchronization mechanisms within OpenSHMEM. Doing this explicitly for all operations and mechanisms provided by OpenSHMEM would be too labor-intensive for the scope of this thesis but would be required for a complete formal model of OpenSHMEM.

4.2.1. Communication semantics

The most significant difference between OpenSHMEM and MPI RMA can be illustrated using the core functionality of both OpenSHMEM and MPI RMA. RMA operations in MPI RMA are non-blocking and need additional calls to ensure origin and target completion, such as `MPI_WIN_UNLOCK_ALL`. However, OpenSHMEM provides two different versions for RMA operations.

The standard versions of all communication routines within OpenSHMEM are *locally blocking*. This indicates that upon return from the routine, all involved buffers are ready to be (re)used. This behavior has some implications for the completion of operations. As for a put operation, this could indicate that the data has either been transferred or buffered, and the operation is, therefore, origin completed upon return. More interestingly, for get operations, this indicates both origin and target completion, as the values must have been fetched from the target to fill the destination buffer of a get operation correctly. This reasoning similarly applies to AMOs as defined by the OpenSHMEM specification. In comparison, the non-blocking versions of OpenSHMEM communication routines are almost identical in functionality and semantics with their MPI RMA counterparts, as their return neither indicates origin nor target completion. Therefore we define the orderings of the new action types in the model to reflect the semantics of OpenSHMEM communication routines.

RMA routines

In the OpenSHMEM specification, the API description for a blocking put operation states that "the routines return after the data has been copied out of the source array on the local PE. The delivery of data words into the data object on the destination PE may occur in any order. Furthermore, two successive put routines may deliver data out of order unless a call to `shmem_fence` is introduced between the two calls." [8, Ch. 9.6.1.1] In comparison, a blocking get routine returns after the data has been delivered to the destination array.

This implies that upon return from a blocking communication routine, the operation has achieved origin completion. Therefore, all memory accesses on the origin must happen while the routine has not returned, and therefore all memory accesses must be consistent with the following accesses. Therefore we define the following orders of actions:

$$(rcp|rcg) \xrightarrow{po} z \implies (rcp|rcg) \xrightarrow{co} z \quad \text{for } z \in \{l*, r*, s*\} \tag{R3}$$

An illustration of this rule is visible in Figure 4.1.

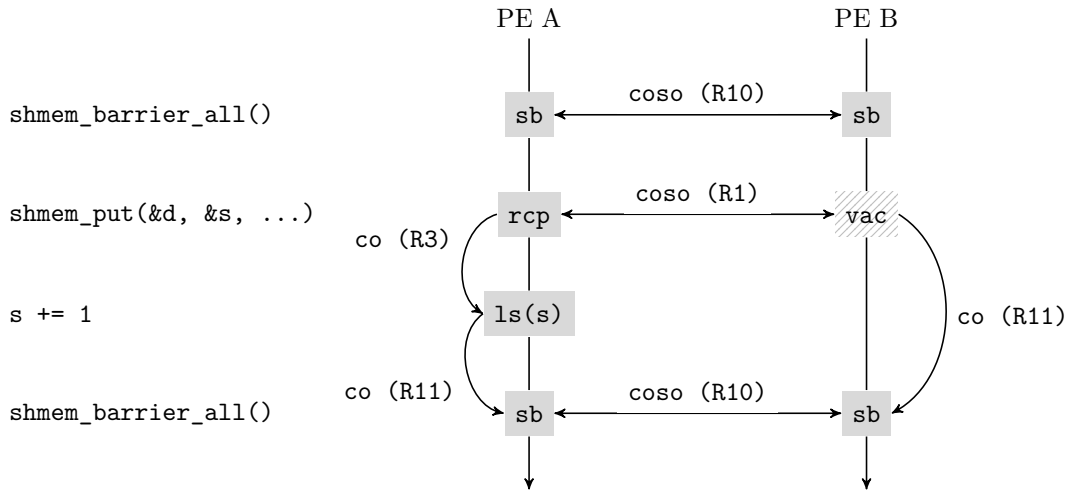


Figure 4.1.: Visulization of the action order for an example execution using `shmem_put`.

In comparison, such guarantees are not given to the non-blocking version of these routines. For `shmem_put_nbi`, the specification states the following:

"The routines return after initiating the operation. After a subsequent call to `shmem_quiet` [This includes any call performing a `shmem_quiet`, such as a `shmem_barrier_all`.], the operation is considered complete. After the completion of `shmem_quiet`, the data has been copied into the dest array on the destination PE. The delivery of data words into the data object on the destination PE may occur in any order." [8, Ch. 9.6.2.1]

Therefore, the memory access on the origin in these routines can only achieve consistency when an OpenSHMEM synchronization call completes them. This opens up the possibility for a data race at the origin, as conflicting memory actions might not be consistent. Figure 4.2 illustrates an example of such a data race, where the source buffer of a non-blocking put operation is modified before the operation is guaranteed to be completed on the origin.

However, there is no ordering guarantee at the target in both the blocking and non-blocking versions of these routines. Therefore the virtual communication actions representing the effects at the target are only ordered in relation to virtual synchronization actions or a `shmem_barrier_all`. As the semantics of the barrier are described later, we only define the following rules here:

$$vac \xrightarrow{hb} vas \implies vac \xrightarrow{co} vas \qquad \text{if } vac.o = vas.o \qquad \text{(R4)}$$

$$vas \xrightarrow{hb} vac \implies vas \xrightarrow{co} vac \qquad \text{if } vac.o = vas.o \qquad \text{(R5)}$$

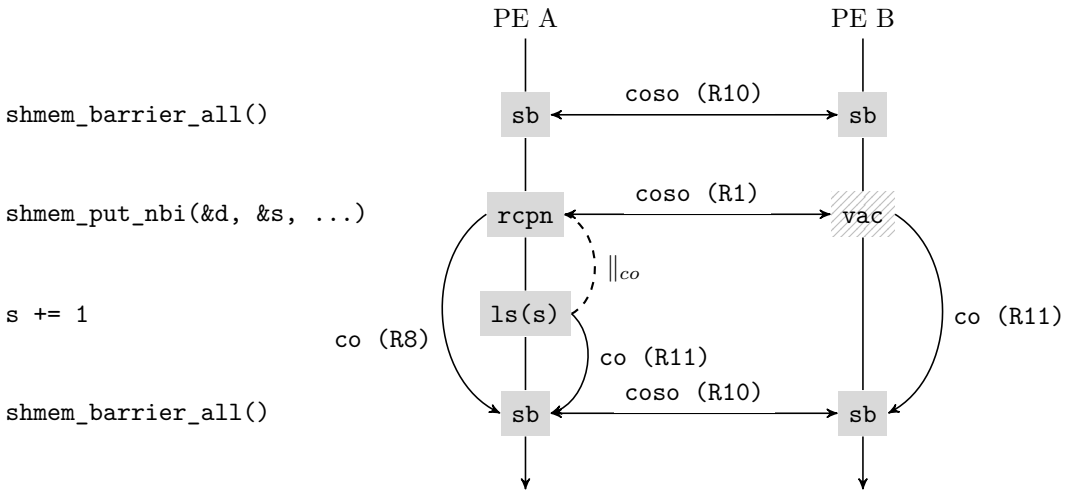


Figure 4.2.: Visulization of the action order for an example execution containing a local buffer race.

4.2.2. Synchronization semantics

Similarly to communication, synchronization also shares many similarities between both models and some significant differences. Many synchronization mechanisms in MPI RMA are semantically equivalent to other synchronization mechanisms in OpenSHMEM. A prime example for this are `shmem_sync` operations which perform process synchronization, just like `MPI_Barrier` synchronizations and `shmem_quiet`, which is equivalent to a `MPI_Win_flush_all` as both perform origin and target completion of previously issued operations.

However, there is also some unique behavior for specific synchronization mechanisms in OpenSHMEM. Specifically, the `shmem_barrier_all` performs process synchronization *and* enforces both origin and target completion, which most resembles `MPI_Win_fence` in active-target synchronization. However, `shmem_fence` is the outlier, as it only enforces the order of operations delivery on symmetric data objects rather than their completion. In MPI RMA, no calls with equivalent or similar semantics exist to a fence in OpenSHMEM.

Memory ordering routine semantics

As fences in OpenSHMEM have no equivalent in MPI RMA, the semantics of this operation must be carefully examined to map them onto this model correctly. In the OpenSHMEM specification document, the `shmem_fence` routine is described to enforce order between the delivery of operations on symmetric data objects. Specifically, it states that "all operations on symmetric data objects issued to a particular PE on the given context prior to the call to `shmem_fence` are guaranteed to be delivered before any subsequent operations on symmetric data objects to the

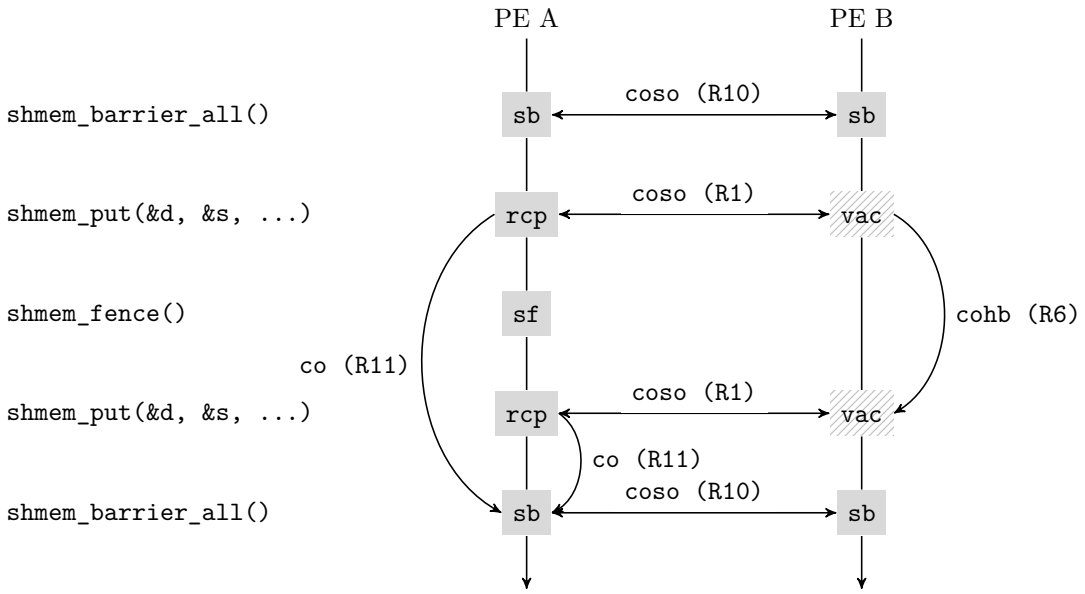


Figure 4.3.: Visulization of the action order for a `shmem_fence()`.

same PE." [8, Ch. 9.11.1] However, it does *not* guarantee completion of the delivery or any order on the delivery of non-blocking get or values fetched by non-blocking AMO routines.

Therefore, our model must reflect this order of deliveries, which are represented by virtual communication actions, and the exemption of values retrieved through non-blocking routines. This is formalized by the following rule and visualized in Figure 4.3:

$$r{*_1} \xrightarrow{po} sf \xrightarrow{po} r{*_2} \implies vac_1 \xrightarrow{cohb} vac_2 \tag{R6}$$
$$\text{if } r *_1 .o = r *_2 .o \wedge r *_1 .t = r *_2 .t \wedge r{*_1}, r{*_2} \notin \{rcgn, raln\}$$

In comparison, the `shmem_quiet` routine "ensures completion of all operations on symmetric data objects issued by the calling PE on the given context." [8, Ch. 9.11.2] This includes the origin and target completion of all outstanding operations issued by the origin PE. This is similar to `MPI_Win_flush_all`, which completes all outstanding RMA operations initiated by the calling process on the specified window. [32] Therefore, we modify the existing definitions by Hoefler et al. [17] and replace the action types with the corresponding OpenSHMEM actions. The resulting rules are given below, with visualization of the action ordering, where `shmem_quiet` prevents a data race, presented in Figure 4.4:

$$r* \xrightarrow{po} sq \implies r* \xrightarrow{co} sq \tag{R7}$$
$$sq \xrightarrow{po} l* \implies sq \xrightarrow{co} l* \tag{R8}$$

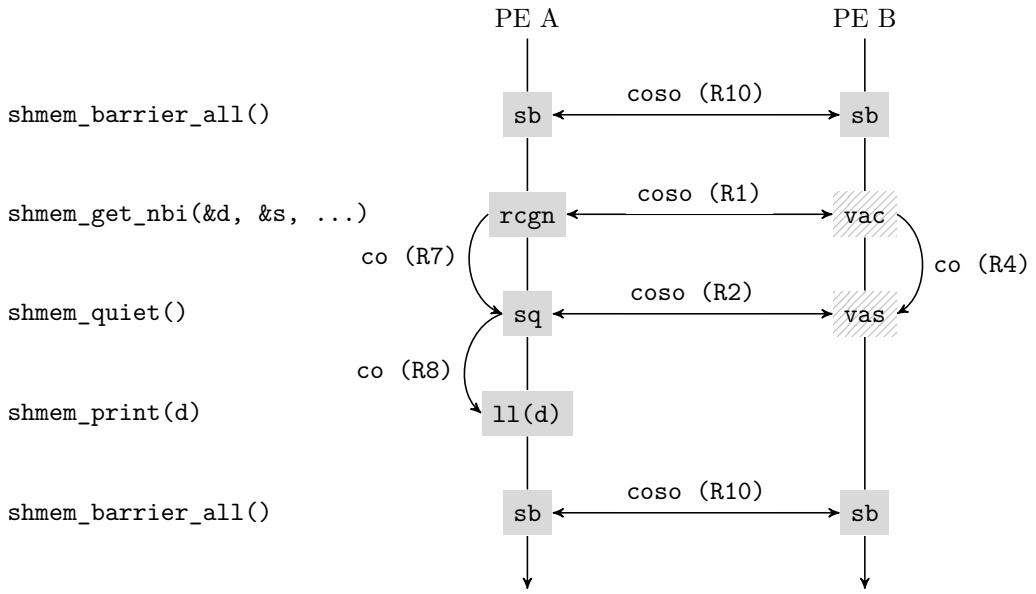


Figure 4.4.: Visulization of the action order for `shmem_quiet()`.

Collective synchronizations semantics

Lastly, we examine the collective synchronization routines provided by OpenSHMEM. As mentioned, these are the `shmem_barrier_all` and the `shmem_sync` routines. The `shmem_sync` routines are the more straightforward mechanism of the two. In its API description, it is stated that "the routine blocks the calling PE until all PEs in the specified team or active set have called *shmem_sync*." [8, Ch. 9.9.3] Therefore, the sync routines in OpenSHMEM perform process synchronization but guarantees no completion of any operations. This indicates that `shmem_sync` routines do not affect the consistency relation between actions and only enforce the happens-before relation. It is equivalent in its semantics to an `MPI_barrier`, and we, therefore, use the exact definition as Hoefler et al. [17] and replace the action type with the types appropriate to OpenSHMEM.

$$ss_i \xrightarrow{so} ss_j \quad \text{for all PEs } \; i, j \in \{0, \ldots, N-1\} \text{ with } N \text{ the number of PEs} \qquad \text{(R9)}$$

Even though the barrier routine in OpenSHMEM shares the name with its counterpart in MPI, they give different guarantees on the state of an application after the processes have returned from the routine. In OpenSHMEM, the barrier indicates complete independence of all operations before and after the barrier as the barrier has to be called by all PEs in the world team collectively and also enforces the completion of all previously issued operations on the default context. Compellingly, the specification explicitly states that "the *shmem_barrier_all* routine is equivalent to calling *shmem_ctx_quiet* on the default context followed by calling *shmem_team_sync*

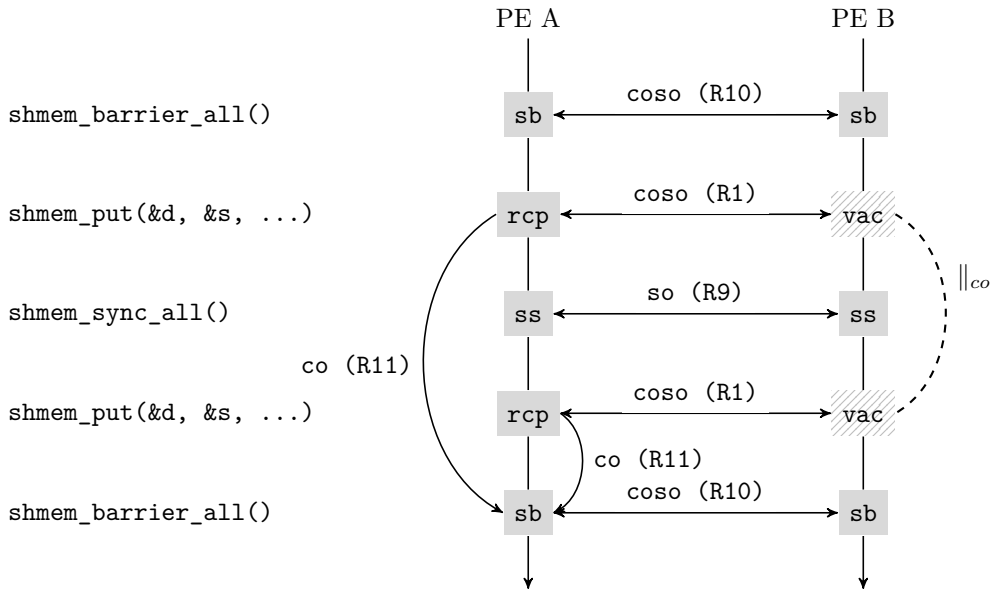


Figure 4.5.: Visualization an action order resulting in a remote race accross a `shmem_sync_all()`.

on the world team." [8, Ch. 9.9.1] We can, therefore, simply combine the rules of sync and quiet routines to achieve an accurate model of the `shmem_barrier_all`, resulting in the following rules:

$$sb_i \xrightarrow{coso} sb_j \quad \text{for all PEs } i,j \in \{0,\dots,n-1\} \tag{R10}$$

$$(r * |l * |vac) \xrightarrow{hb} sb \implies (r * |l * |vac) \xrightarrow{co} sb \tag{R11}$$

$$sb \xrightarrow{hb} (r * |l * |vac) \implies sb \xrightarrow{co} (r * |l * |vac) \tag{R12}$$

Remaining OpenSHMEM features

The communication and synchronization mechanisms described and formalized above are the most prominent within the programming model. However, OpenSHMEM provides many more features, which we could have analyzed in this section. This is mainly due to the complexity of this analysis and the additional information required to model these features. The prime example of this are the previously mentioned AMOs. In OpenSHMEM, for two atomic operations to be guaranteed exclusivity, it is required that both operations are in the same atomicity domain, as explained in Section 2.2.2. However, to model this in this model would require modeling the datatype, the alignment of the memory accessed, and the context used in the operation. All of this is currently not accounted for in this model and would increase its complexity while reducing the readability and intelligibility of the model. Similar additions would be necessary to model communication contexts,

point-to-point synchronization, distributed locks, signaling operations, and collcetive communication routines.

Since the model in this thesis is intended to be a means to an end, we decided to not fully model OpenSHMEM and instead focus and modeling the central concepts and operations within it, this being the RMA operations and synchronization mechanisms. With this, the reader can understand the model and one-sided communications, which is required to understand the data race detection approach we use in our implementation for race detection.

4.3. On-the-Fly race detection

The formal model of the OpenSHMEM semantics developed for this thesis now forms the basis of our data race detection approach. As was defined in Section 4.1.2, a data race can occur when both the consistency and happens-before relation do not order two conflicting operations. Therefore, we must examine a program for these relations to detect data races.

For detecting data races, several common techniques are employed. A survey by Raza [40] categorizes these techniques by on-the-fly, ahead-of-time, and post-mortem techniques. These techniques have advantages and drawbacks and can be combined into hybrid approaches. However, as one of the central premises of this thesis is the similarity of OpenSHMEM and MPI RMA, our data race detection approach is based on previous work regarding MPI RMA. Specifically, the On-the-Fly data race detection for MPI RMA programs in MUST [44, 45] is going to form the basis of the detection implemented in this thesis.

On-the-fly approaches for data race detection use dynamic program analysis. They detect races at run time and only cover feasible execution paths. This has the downside that the analysis only covers some possible execution paths, and the race detection may depend on external factors, such as input and scheduling, instead of purely the program to be analyzed. This restriction can result in false negative results, as some paths execution paths containing races may never be examined, and additionally, the overhead from run time tracking may prevent schedules that could otherwise contain races [40].

In comparison, static techniques such as the static analyzers OpenSHMEM-Checker [5] and OpenSHMEM-Analyzer [14, 37] use an abstraction of the program and exhaustive searches on these abstractions to detect data races and other errors. However, to prevent a complexity explosion in the internal abstraction of the program, the amount of information for a single execution is often reduced compared to on-the-fly or post-mortem techniques, which increases the risk of false positives [40]. However, at the time of writing, these tools do not focus on data race detection and primarily focus on other errors, such as incorrect use of the API, and are, therefore, no direct comparison to our work.

The previous work for data race detection in MPI RMA that we are using as our basis uses the happens-before and consistency relations from the model by Hoefler et

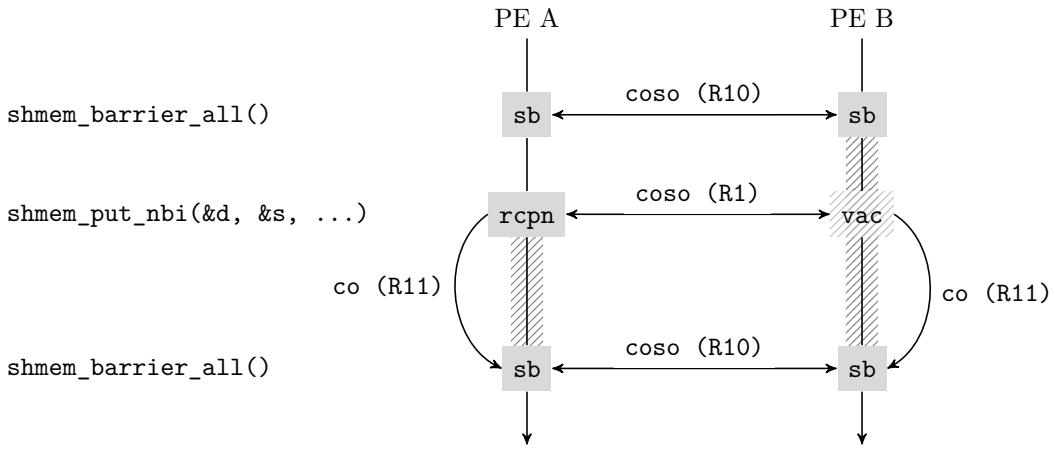


Figure 4.6.: Visualization of the concurrent regions for a non-blocking put operation.

al. to define *concurrent regions* of an RMA operation. For this, an RMA operation is split into two separate operations called the *origin operation* and *target operation*, which represent the local buffer access on the origin and the remote access on the target, respectively. These are approximately the same as memory actions on the origin and virtual communication actions on the target. However, this change in terminology is beneficial when talking about the data race detection approaches most of the time, as we need to refer to actions that are on the same process, which is a narrower perspective than an entire execution.

Under these conditions, concurrent regions are defined by the earliest and latest point in logical time of the memory accesses of an origin or target operations that can take place [45]. Correspondingly, for each RMA operation, there is a concurrent region at both the origin and target of the RMA operation. Figure 4.6 visualizes these regions for a non-blocking put operation. A definition of these regions is given in Section 4.3. These concurrent regions catch additional information necessary for the existing shared-memory data race detection tools to grasp the effects of RMA operations. In order to determine these concurrent regions for each RMA operation, the happens-before relation has to be tracked in the execution of a program. The *happens-before order* ($\xrightarrow{hb}$) is the transitive closure of the *program* ($\xrightarrow{po}$) and *synchronization* ($\xrightarrow{so}$) orders as described in Section 4.1.

The program order is defined using the order in which statements of the program on a process are executed. However, capturing the synchronization order requires additional effort. It requires exchanging information about synchronization calls between involved processes. The initial intuition for capturing the ordering of operations may be to record the wallclock time when an operation is performed. These timestamps could then be compared, and the order of two arbitrary operations established. However, this has two requirements for the timestamps to establish a correct ordering between operations: First, the clocks on the processes must be precise enough to differentiate between individual operations and second be synchronized to

have a common reference frame. These requirements are challenging to realize in a distributed system [10].

Vector clocks

Fortunately, this is not a new problem and has been researched extensively. In a paper, Lamport [21] defines another *happened-before* relation. For this relation, Lamport assumes that each process consists of a sequence of events. Thus a process is defined as a set of events with an a priori *total ordering*. This ordering can then be extended by defining the happened-before relation $\rightarrow$ as the smallest relation which satisfies three conditions:

1. If two events a, b are on the same process and a *comes before* b in the process, then $a \rightarrow b$.

2. If a is the sending of a message on one process and b is the receiving of the *same* message on another process, then $a \rightarrow b$.

3. $\rightarrow$ is transivitve.

This captures the total event ordering, or in terms of our formal model, the *total action ordering* of an execution. However, this ordering is only one of the many possible valid event orderings for a particular distributed computation. However, to detect data races efficiently, it is desirable to capture all possible interleavings in a particular execution in a partial order to determine which actions are concurrent. Thus the earliest and the latest possible points in logical time when an action is executed are needed.

For such a partial ordering, Fidge [10] and Schwarz and Mattern [43] separately introduced the concept of *vector clocks*. Vector clocks allow the capture of the *happens-before relation* as a strict partial order that matches the definition in the semi-formal model of OpenSHMEM and MPI RMA presented in Section 4.1.1. The general concept behind vector clocks can be applied to many programming models and is widely used in correctness analysis tools. In short, a *vector clock* is an array

$$\mathcal{V} := [c_0, c_1, \ldots, c_{n-1}]$$

of integers with one value c_i for each process i. Each process $p \in \{0, \ldots, n-1\}$ maintains this a local copy $\mathcal{V}_p$ of this array and allows it to have local knowledge about the synchronization with all other processes [10]. Within this local copy of process p, we call the entry at position $\mathcal{V}_p[p]$ the *local clock value*. For a program launched with three processes, a vector clock array on process 1 may have the values $\mathcal{V}_1 = [3, 4, 1]$ with $\mathcal{V}_1[1] = 4$ being the local clock value for process 1. All other values in a local copy that are not the local clock value represent the clock values from the other processes, or in other words, the clock values at the last known synchronization points.

The vector clocks used in this thesis are from the previous work on MPI RMA [45, 44]. They have been adapted from the work by Schwarz and Mattern [43]. The vector clocks for all processes are zero-initialized and updated based on rules when processes execute an action.

Definition 1 (Vector Clock Update Rules [44])**.** Let $P_0, \ldots, P_{n-1}$ denote the processes of a distributed computation. The vector clock $\mathcal{V}_i$ of process P_i is maintained according to the following rules:

1. Initially, $\mathcal{V}_i[k] := 0$ for all $k \in \{0, \ldots, n-1\}$
2. When proccess P_p executes an action, the local clock value $\mathcal{V}_p[p]$ is incremented by one: $\mathcal{V}_p[p] := \mathcal{V}_p[p] + 1$.
3. When process P_p sends in an action a signal to process P_q, it sends its own vector clock $\mathcal{V}_p$ to process P_q.
4. When process P_p waits in an action for the signal of another process P_q, after finishing waiting, it merges its own vector clock $\mathcal{V}_p$ with the received vector clock $\mathcal{V}_q$ in an element-wise maximum operation:

$$\mathcal{V}_p[i] := max\left(\mathcal{V}_p[i], \mathcal{V}_q[i]\right)$$

These generic rules need to be applied to the programming model and match the semantics of the different features within the programming model. For example, in a `shmem_barrier`, all processes are waiting on each other to arrive in the routine. This is equivalent to all processes sending messages to all remote processes and waiting for messages from all remote processes to arrive. Therefore after a `shmem_barrier`, all processes are synchronized with the same vector clock values. This mapping of OpenSHMEM has already been done in research on a generic vector clock module for the MUST framework [46].

Under these rules and a correct mapping of the programming model, all processes should have accurate information about their synchronization with other processes in the vector clocks. The vector clocks $\mathcal{V}_p, \mathcal{V}_q$ of processes P_p and P_q can then be compared using the following relation [46]:

1. $\mathcal{V}_p \leq \mathcal{V}_q$ iff $\mathcal{V}_p[i] \leq \mathcal{V}_q[i]$ for $i \in \{0, \ldots, n-1\}$
2. $\mathcal{V}_p < \mathcal{V}_q$ iff $\mathcal{V}_p \leq \mathcal{V}_q$ and $\mathcal{V}_p \neq \mathcal{V}_q$

Schwarz and Mattern [43] proposed and proved a theorem that allows reasoning about the happens-before as defined in our model by using vector clocks. For two actions $a, b \in A$ a executed by two processes p and q *happens-before* b if and only if $\mathcal{V}(a)_p < \mathcal{V}(b)_q$ where $\mathcal{V}(a)_p$ is the vector clock of process p at the execution time of action a. Therefore, using the existing vector clock module, we can calculate the happens-before relation between individual actions.

Concurrent intervals of actions

Moreover, vector clocks can additionally be used to identify the interval in logical time in which an individual action a is concurrent ($\|_{hb}$) to actions at other processes. These intervals are identified by the concept of the *last signal* and *next wait*.

Definition 2 (Last Signal [44])**.** Let $a \in A$ be executed by process p. The *last signal* from process q to process p happening before a, denoted by $LS_{q\to p}(a)$, is a synchronization action $l \in A$ on process q where

$$\mathcal{V}(l)_q[q] = \mathcal{V}(a)_p[q].$$

Definition 3 (Next Wait [44])**.** Let $a \in A$ be executed by process p. The *next wait* from process q for process p happening before a, denoted by $NW_{q\to p}(a)$, is the synchronization action $n \in A$ on process q where

$$\mathcal{V}(n)_q[q] \geq \mathcal{V}(a)_p[p]$$

and

$$\mathcal{V}(n)_q[q] \leq \mathcal{V}(n')_q[q] \text{ for all } n' \in A \text{ with } \mathcal{V}(n')_q[p] \geq \mathcal{V}(a)_p[p].$$

In the work that introduced these concepts [44], the author also provides an intuition where the last signal $LS_{q\to p}(a)$ is described as the latest action in logical time that happens before a on process q, and the next wait $NW_{q\to p}(a)$ is the earliest action on process q that happens after action a on process p.

Using the concepts of the last signal and next wait, we can define the *concurrent interval* of each action a executed on process p [44]. Following the definitions given above, each action has one corresponding concurrent interval for the action a at each process $q \neq p$ with the lower bound being the last signal ($LS_{q\to p}(a)$) and the upper bound the next wait ($NW_{q\to p}(a)$). This interval in logical time is open. Therefore, the last signal and next wait actions are not concurrent to the action a. This relation more intuitive from the viewpoint of process q, where for all actions $b \in A$ executed by process q with

$$LS_{q\to p}(a) \xrightarrow{po} b \xrightarrow{po} NW_{q\to p}(a)$$

it follows that $a \|_{hb} b$.

Additionally, for the edge cases with no preceding last signal or trailing next wait, the concurrent interval begins at and including the first action and ends on including the last action executed on the process. Figure 4.7 shows the concurrent intervals for an action $w \in A$ executed by process 3.

Concurrent regions of an RMA operation

We can now define the *concurrent regions* for a remote communication action. The concurrent regions for an RMA operation capture all actions that could result in a data race if they are conflicting. As two conflicting actions x and y need to be

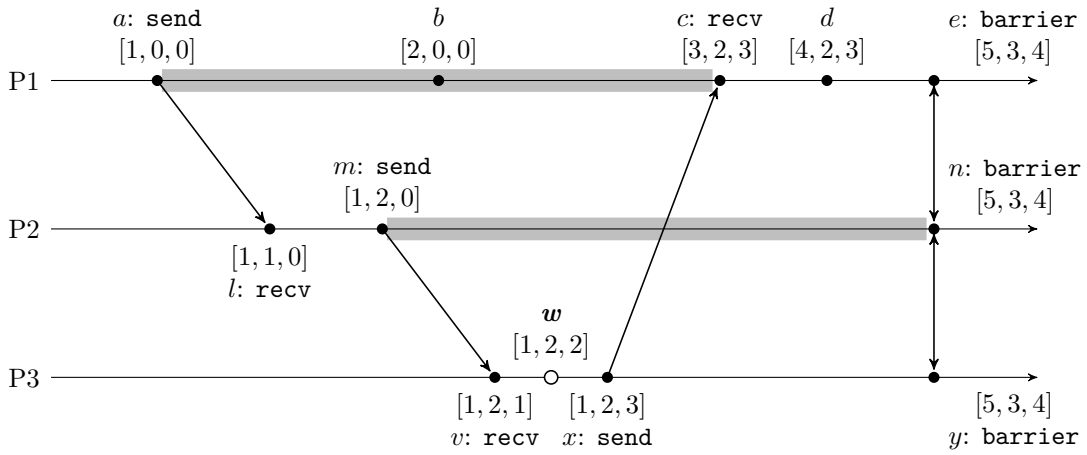


Figure 4.7.: Concurrent interval example of an action $w \in A$ at process 3. The concurrent intervals of w at processes 1 and 2 are marked gray. [44]

ordered according to $x \xrightarrow{cohb} y \vee y \xrightarrow{cohb} x$, both the consistency and happens-before order need to be considered for the concurrent region.

An RMA operation affects only the origin and target of the operation. Thus, each RMA operation has two concurrent regions, one at the origin and one at the target. At the origin, the local buffer access of the operation is represented using the remote communication action itself. For example, a remote communication put (*rcp*) represents a local load at the origin. The communication routine itself is the start of its concurrent region, while the routine that enforces the consistency order on the origin or, in simpler terms, completes the RMA operation *locally* is the end of the concurrent region. For the standard blocking put, this would be the communication routine itself as it is *locally blocking*. However, for a non-blocking put, an additional synchronization routine, such as `shmem_quiet`, would be necessary. This additional routine enforces the consistency order, while the program order gives the happens-before order. Figure 4.8 shows the concurrent regions for such a blocking put operation.

The concurrent region, which encompasses the virtual communication action (*vac*) of an RMA operation, is more complex. The virtual communication represents the effects of an RMA operation at the target, e.g., a local store for a put operation. However, in the case of one-sided communication and synchronization, the target process cannot use the program order to infer the happens-before relation. As these orders can be enforced separately by synchronization calls in OpenSHMEM, e.g., `shmem_quiet` and `shmem_sync_all`, we cannot simply use the last signal and next wait of the remote memory action as the concurrent region of the virtual memory action. The formal model for OpenSHMEM provided in this chapter provides more insight into the virtual communication action.

Specifically, assume we have two actions so, st with $so \xrightarrow{so} st$ and a remote communication action r where so and r are executed on the origin and st and the

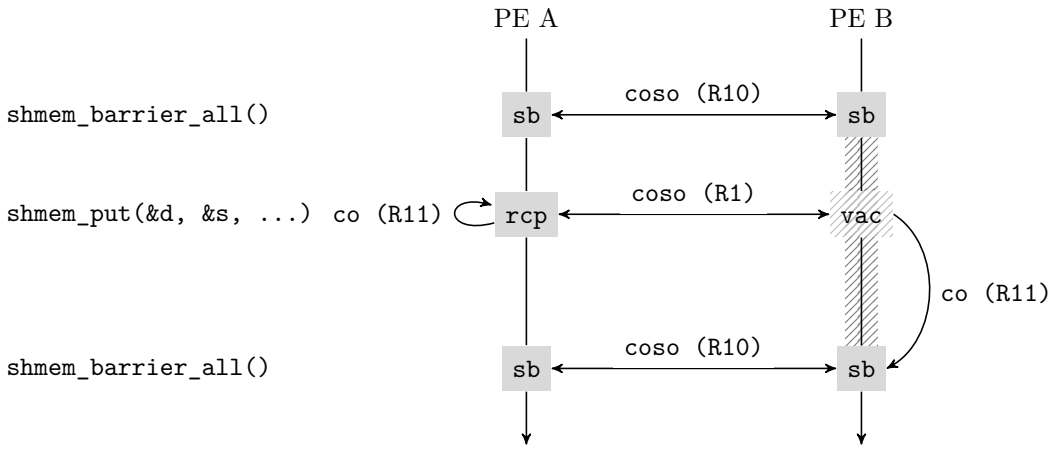


Figure 4.8.: Visualization of the concurrent regions for a blocking put operation. The concurrent region at the target of the put operation is marked using hte hatched area.

virtual communication action vac of r are executed on the target. Then it follows that from Rule R1 that the corresponding virtual communication action vac happens after st or simply $st \xrightarrow{hb} vac$. Therefore, from Definition 2, it follows that the last signal of the remote communication action r must be the earliest possible time the effects of r can take place at the target as shown in Figure 4.8.

However, in OpenSHMEM as in MPI, process synchronization can be achieved without enforcing the completion of issued operations by routines like `shmem_sync_all`. Therefore we cannot rely on the next wait of the remote communication action as $vac \xrightarrow{hb} NW_{o\to t}(r) \implies vac \xrightarrow{co} NW_{o\to t}(r)$. However, vac will happen before and be consistent with the effects at the target of a completing routine such `shmem_quiet` (sq) and its corresponding virtual synchronization action (vas) or `shmem_barrier_all` (sb) or in short $vac \xrightarrow{cohb} (vas|sb)$. Therefore, the latest possible point in logical time when the RMA operation completes at the target is the latest possible time when such a completing synchronization occurs or simply the next wait of the completing action $NW_{o\to t}((sq|sb))$.

Therefore, we define the concurrent region at the target of an RMA operation like Schwitanski et al. [45]:

Definition 4. Let $a \in A_o$ be the action corresponding to any RMA operation issued as the origin process P_o to the target process P_t. Additionally, let $c \in A_o$ be the synchronization action issued at P_o guaranteeing target completion. The concurrent region of the remote access at P_t is $(LS_{t\to o}(a), NW_{t\to o}(c))$.

Therefore to determine the concurrent region at the target of the put operation in Figure 4.8, we need to identify the last signal of the rcp action and the next wait of the completing action. The last signal in the example is simply the last action where

process synchronization occurred or, in this case, the initial barrier. Similarly, the following barrier is equivalent to calling `shmem_quiet` on all PEs and performing a collective `shmem_sync_all` call afterward. Thus, the completing action is the barrier, and it is simultaneously the next point of process synchronization between both PEs, resulting in the concurrent region marked in Figure 4.8.

5. OpenSHMEM data race detection in MUST

In the previous chapters of this thesis, we developed an understanding of OpenSHMEM and its semantics, especially of the RMA operations and synchronization mechanism within it. Additionally, we defined data races and adapted a formal model of MPI RMA to OpenSHMEM.

The formal model provides essential insights into the semantics of OpenSHMEM and clarifies the interactions of RMA routines, Atomic Memory operations, and synchronization collectives and routines. However, while the formal model might aid developers with comprehending the programming model and possible pitfalls, tooling which can understand and detect errors such as data races in existing arbitrary codes has become indispensable during application development.

Developing such a tool, to at least a prototype stage, is part of the expressed goals of this thesis. To accomplish this, as previously mentioned, we will build on existing work for data race detection in MPI RMA [45, 44]. This race detection for MPI RMA is implemented as a module in MUST (Marmot Umpire Scalable Tool) [16], which is an event-based runtime correctness-checking utility for MPI applications. However, recently, additional programming models have been supported within it. Among these additional models is OpenSHMEM, as a generic distributed vector clock module has already been implemented which supports the process synchronization mechanisms within OpenSHMEM [51, 46]. Therefore, the initial groundwork for OpenSHMEM race detection is already completed, reducing the development effort for code that is not directly involved in data race detection.

5.1. Existing infrastructure for MPI RMA

The general framework in which we will implement our race detection is MUST [16], which is built on several components such as the widely used P^NMPI [41, 42] and the Generic Tool Infrastructure (GTI) [15], to implement its functionality and enable features used by modules within MUST. Additionally, we will use the existing data race detection tool ThreadSanitizer (TSan) [48, 49] to perform the bulk of the data race detection. Knowledge of all these tools and their features is required to understand our implementation and design.

5.1.1. MUST: Marmot Umpire Scalable Tool

MUST or the *Marmot Umpire Support Tool* [16] started as an effort to create a runtime error detection tool for MPI applications based on experience gained from the *Marmot* [20] and *Umpire* [53] correctness checking tools. During the design of MUST, some of the central objectives were the creation of a reliable tool that does not report false positives while also achieving scalability such that efficient analysis of applications using a large number of processes is feasible, as this was a limiting factor in the previous tools [16].

MUST uses error detection at runtime with different places where error analyses can run to achieve these goals. While all error detection mechanisms are achievable using global knowledge during runtime, sharing all information globally inhibits the scalability of the error detection and is not required for all analyses. For example, detecting incorrect parameters passed to an MPI call might only require knowledge local to the rank, while deadlock detection using a graph analysis approach requires global knowledge. Therefore, MUST implements an architecture where modules that perform some of the analyses can run on a specified *place* depending on its requirements. Figure 5.1 illustrates these places and on which places specific analyses run.

This architecture is implemented in three layers. The lowest layer builds on another tool or rather an extension to the profiling interface for MPI called P^{N}MPI . It allows several tools that use the PMPI profiling interface to run simultaneously. This allows MUST modules to access the parameters of an MPI call as if it were directly accessing the profiling interface. It also allows MUST to implement features such as rewriting MPI communicators, which requires modifying parameters passed to the MPI library.

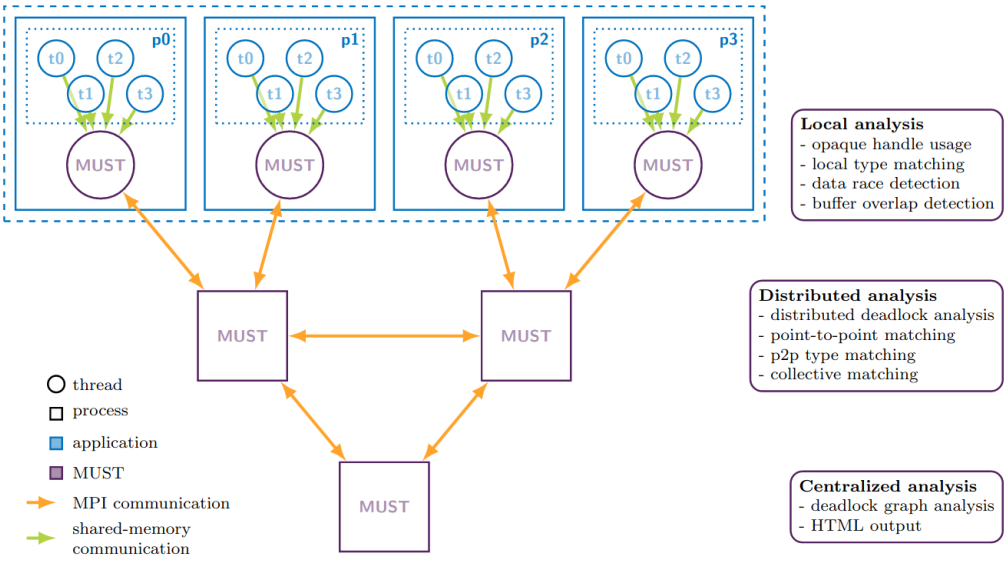


Figure 5.1.: Analysis placement in MUST. [39]

The top layer consists of the correctness checks, also implemented as modules. These correctness checks require information ranging from the parameters of MPI calls and information that may not be available on the local place. To facilitate the exchange of such information, MUST relies on another project called *Generic Tool Infrastructure* (GTI) [16, 15]. GTI aims to provide tool developers with a portable and generic infrastructure for runtime error detection tools in distributed applications using MPI. It allows the exchange of information between places and the offload of correctness checks into places separate from application processes and threads.

A tool built on GTI is implemented in modules that can be configured into a tool instance using a variety of specifications in the XML format. In these specification files, the interfaces of analysis functions, mapping of analysis functions to MPI calls, and other configurations can be defined. Specifically, one configuration file is used to declare the number of tool threads and processes and their communication methods. Additionally, a tool developer can choose the place where an analysis executes. If the analysis occurs within the application process, the analysis is called a *local analysis*. In contrast, if it takes place on a separate tool thread or process, it is called a *tool analysis*. The local analyses running directly under P^{N}MPI form the previously mentioned lowest layer in MUST, while tool analyses form the upper layer. The interactions of these different components in a GTI-based tool are illustrated in Figure 5.2.

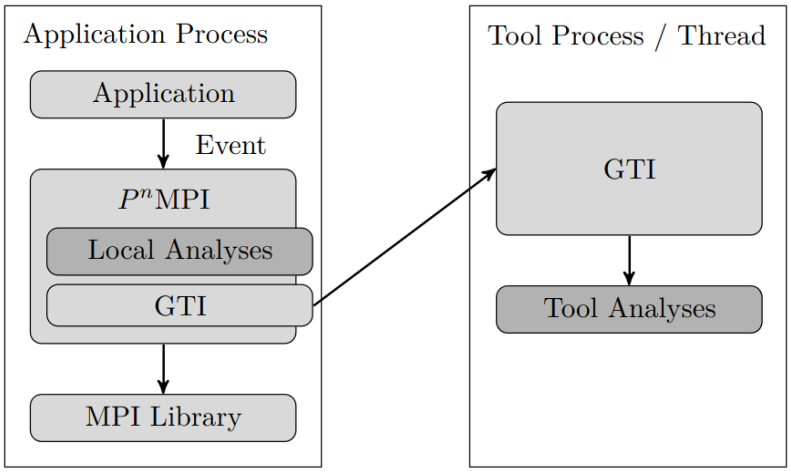


Figure 5.2.: Tool using GTI. [15]

5.1.2. P^{N}SHMEM

So far, the previously mentioned tools have all been based on MPI with its simple profiling interface to facilitate tools' portability. Otherwise, tools might have to rely on platform-specific mechanisms to attach to an application or use other forms of instrumentation. Similarly, P^{N}MPI only provides a virtualization layer for the MPI profiling interface. However, the goal of this thesis is to develop a data race detector

for OpenSHMEM, so we cannot use P^{N}MPI directly. However, OpenSHMEM provides a similar profiling interface, which has proven indispensable for MPI tool development.

The OpenSHMEM profiling interface works similarly to its MPI counterpart. The entry point can be replaced for every OpenSHMEM function using the `shmem_` prefix and tool code can execute instead of executing library code. The original library function can still be accessed using the function name with the `p` prefix.

However, as MUST relies on P^{N}MPI and it does not support OpenSHMEM, such functionality must either be added to P^{N}MPI or otherwise transparently provided to MUST. Previous work on a generic vector clock module for distributed-memory programs was implemented in MUST [51, 46]. This module already supports OpenSHMEM and is implemented using a patched version of P^{N}MPI that can provide the same functionality for OpenSHMEM. This patched version, called P^{N}SHMEM internally, is already integrated into some development branches of MUST and GTI.

5.1.3. TSan: ThreadSanitizer

ThreadSanitizer (TSan) [48, 49] is an on-the-fly data race detector for shared-memory programs. TSan went through two iterations of its design. Initially, it tracked memory accesses and synchronization of threads by using the dynamic binary instrumentation of Valgrind. However, this incurred a significant slowdown with a factor of 20 to 300 [50]. Its current version uses compiler instrumentation to perform a data race analysis. It is available in the LLVM/Clang [27] and GCC [26] projects as a sanitizer similar to *Address Sanitizer* [47] through the compiler flag `-fsanitize=thread`. This newer design reduces the slowdown to a factor of 2-10.

This reduction in the slowdown is achieved by instrumenting each memory access, synchronization, and function call using special functions. Code 5.1 illustrates this instrumentation. The instrumentation of memory accesses and synchronization is required to feed the happens-before analysis, while the annotation of functions is required to correctly print the stack trace when a data race is found. This annotation of function calls is required since TSan must print two stack traces, one for each conflicting memory access. Printing the stack frame is simple for the access during which TSan detected a data race, as the current stack trace is immediately available. However, the stack trace from "previous" access on which TSan had not yet detected a race is not available from the current stack. Therefore TSan needs to gather this additional information.

To efficiently track memory accesses and detect data races without much overhead, TSan uses a concept called *shadow memory*. In short, TSan splits the virtual address space into two separate regions, the application memory and the shadow memory. The application memory is the memory used by the instrumented application, and each 8-byte word is directly mapped to N 8-byte words of shadow memory called the *shadow state* of that 8-byte application word. Each shadow word represents a memory access and contains information on the thread, epoch (logical access time),

```
void someFunction ( int * p ) {
    *p = 12;
}
```

```
void someFunction ( int * p ) {
    __tsan_func_entry(__builtin_return_address(0));
    __tsan_write4(p);
    *p = 12;
    __tsan_func_exit();
}
```

Code 5.1.: Compiler instrumentation of TSan for a function setting a value. [12]

access size, access offset, and whether the access is a write. The layout of a shadow word is shown in Figure 5.3.

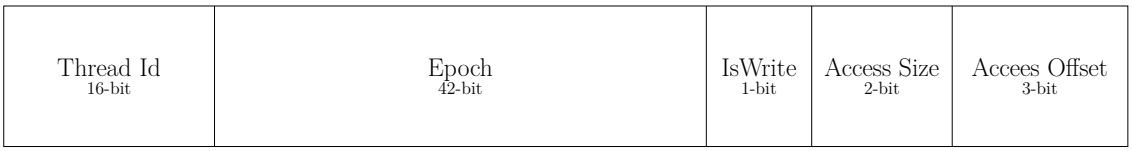


Figure 5.3.: Layout of a shadow word. [12]

Most of this information is easily obtained through lookups or simple computations. However, the epoch requires more effort. To track the epoch TSan internally maintains a vector clock, similar to the one in MUST [51, 46], in thread-local storage for each thread. This vector clock is used to track synchronization between threads in the application.

The actual data race detection in TSan is done by its internal state machine each time a new memory access is tracked. The TSan documentation [12] describes the data race detection and its internal state machine in detail. However, a general understanding of TSan's race detection mechanism can be gained from previous work on data race detection in MPI RMA. It provides a general description of the state machine update process on a new memory access [44]:

1. Increment the local vector clock value for the thread making the access.
2. Create the new shadow word for the memory access.
3. Iterate over all shadow words in the shadow state for the accessed application word and check for the following conditions against the new shadow word:
   a) Mismatched thread IDs.
   b) Overlapping accesses.
   c) At least one of the two "IsWrite" bits is set.
   d) The accesses are not happens-before ordered.

If all these conditions are met for a shadow word in the shadow state and the new access, then TSan detects a race.

4. Insert the new shadow word into the shadow state. The new word is placed preferentially into an empty spot. If there is no empty spot, TSan tries to replace a shadow word that happened before the new word, or if this is also not possible, then TSan replaces a random shadow word.

Due to this approach of using limited shadow memory and compiler instrumentation, TSan has a few limitations. In some situations, TSan might have already evicted potentially conflicting memory accesses from the shadow state and therefore misses a race, and the random eviction policy for concurrent accesses results in a non-deterministic analysis. Additionally, the requirements for compiler instrumentation prevents data race detection in prebuilt binaries or when an application uses inline assembly, and TSan might not understand some constructs, such as application-specific constructs like mutexes.

However, TSan provides an interface [48] for users to manually annotate their code to mitigate some of the latter problems. An example is the ability to annotate *mutual exclusion* in the application to prevent TSan from reporting false positives. This is achievable using the `ANNOTATE_HAPPENS_BEFORE(ptr)` and `ANNOTATE_HAPPENS_AFTER(ptr)` annotation functions.

Over time and the iterations of TSan, these annotations have been extended to include a concept called *fibers* [29, 28]. Fibers represent additional concurrency units within a thread and can be used to provide information on concurrent events in a thread. They can be dynamically allocated and annotated during the execution of a program. For example, a non-blocking `MPI_Irecv` writes to memory without the knowledge of the application, and an application can only check whether the access has already been completed. A tool can therefore use fibers to annotate the write by the uninstrumented MPI library run-time in a fiber and use TSan to check for data races involving MPI calls.

5.1.4. RMA operation tracking in MUST

This software stack, presented in the previous sections, was used by Schwitanski et al. to develop a data race analysis for MPI RMA programs in MUST [44, 45]. This data race detection implementation is split into several modules which encapsulate different functionality of the RMA race detection and additionally uses several pre-existing features within MUST, such as MPI communicator and datatype tracking. However, the central module of this analysis is called *RMATrack* in the analysis group *OneSidedChecks*. An overview of the general architecture for the data race analysis is illustrated in Figure 5.4.

RMATrack and some other modules involved in MPI RMA race detection in MUST are offloaded to a tool thread on each process separate from the application threads since local analyses cannot exchange information between processes through

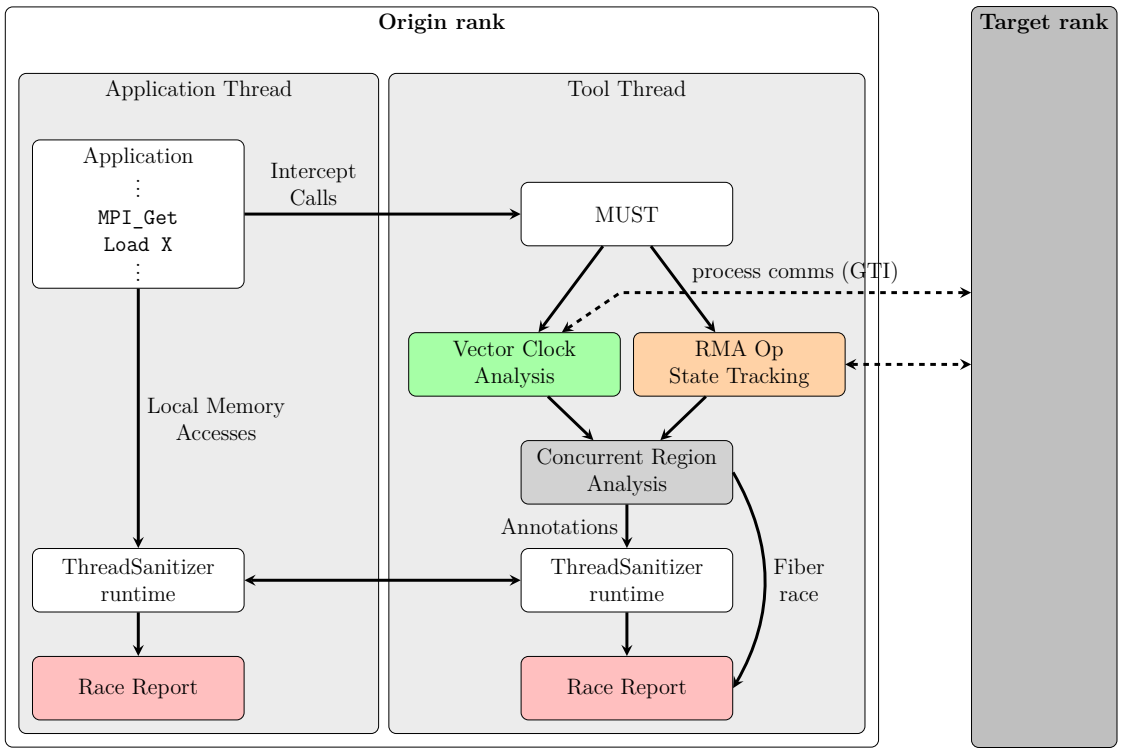


Figure 5.4.: Analysis architecture for detecting MPI RMA data races using RMATrack in MUST, adapted from [45].

GTI. Exchanging information between processes is only possible at the tool thread level and above. This exchange of information is required for both the vector clock module and information on the RMA calls at the target such that RMATrack has all information necessary to perform its race detection.

The race analysis in MUST uses TSan as its central race detector to facilitate efficient data race detection and to instrument, local memory accesses such as loads, stores, or common functions like `memcpy`. However, as mentioned in Section 5.1.3, TSan is only aware of memory accesses by instrumented code. The application's local accesses are instrumented by enabling TSan during the compilation of the application, but memory accesses by non-instrumented code, like the MPI library, will not be registered.

Providing these annotations to TSan is the central task of RMATrack, and it is achieved by intercepting RMA communications and synchronization calls using P^NMPI and passing the necessary information up to the tool thread. Each RMA communication call is split into an *origin operation* and a *target operation*. At the origin of an RMA communication call, RMATrack has local knowledge of the call and can store it in an origin operation. At the target of an RMA communication call, the process does not know the communication and its corresponding memory accesses due to the one-sided nature of the call. Therefore, RMATrack passes the information on the call to the target of the operation, where the RMATrack instance

receives said information and stores it in as a target operation. Both origin and target operations are tracked until RMATrack can calculate their concurrent regions, after which all operations are annotated such that TSan can perform the data race analysis.

These annotations are then performed when the concurrent region for an operation ends, that is, its *next-wait* is encountered. Each operation is annotated on a fiber of the tool thread, where each process of the execution corresponds to a fiber that is created during the program launch. For each annotation of a memory access RMATrack switches to the corresponding fiber of the origin of the operation. The sync clock (internal vector clock of TSan) value is then set to the value of the last signal of the operation. The memory access is then annotated, after which the fiber is switched back. This approach has one complication. For the TSan race detection, algorithm fibers are treated like any other thread. Specifically, accesses annotated on the same fiber are still ordered. Thus, accesses on the same fiber will result in a data race according to the TSan runtime as it increments the internal clock value for the fiber after each annotated access. Since all target operations coming from the same origin are annotated on the same fiber, a manual race analysis for these operations is required to detect the data races of these operations correctly. Therefore, each time a target operation is received, it is checked against all other concurrent target operations that were previously received.

With this procedure, RMATrack annotates each RMA operation. However, it needs to know the concurrent region of each operation. Thus RMATrack also tracks the synchronization mechanisms between processes. For this, it uses the existing vector clock module to track the process synchronization $\xrightarrow{hb}$ between processes, as it is assumed that the application is single-threaded, but the consistency relation $\xrightarrow{co}$ is tracked using the RMATrack module.

5.2. Implementation of the OpenSHMEM race detection

For the data race detection in OpenSHMEM programs we implement in this thesis, we adapted the existing race detection in MUST for MPI RMA [45, 44]. Since OpenSHMEM and MPI RMA are semantically similar [13], we try to reuse much of the existing infrastructure. Reusing existing infrastructure has some advantages, mainly the lower development effort and the reduction of possible bugs and similar problems. In general, for an OpenSHMEM data race detection analysis, the core of this work is to develop the "front-end" of the analysis and accurately map the OpenSHMEM semantics from the formal model in Chapter 4 to the existing race detection.

Our implementation, therefore, modifies or adds the following to the existing MPI RMA race detection:

1. Extending the existing OpenSHMEM specification in MUST with all RMA communication and synchronization calls such that MUST is aware of the entire OpenSHMEM 1.5 specification. Previously, MUST was only aware of a subset of the OpenSHMEM 1.4 specification required for the vector clock module.

2. Add an overarching module responsible for intercepting and tracking OpenSHMEM RMA and synchronization calls which we call *ShmemTrack*.

3. Add analysis functions that capture all the necessary information from the OpenSHMEM calls and makes them available to ShmemTrack.

4. Add tracking of OpenSHMEM allocations to annotated memory accesses on dynamically allocated symmetric memory correctly.

5. Add the unique synchronization mechanism `shmem_fence`, which has no direct equivalent in MPI RMA.

6. Properly map the analysis functions from ShmemTrack to OpenSHMEM according to the semantics defined in the formal model of OpenSHMEM. For example, correctly mapping `shmem_malloc` requires modeling both tracking the performed allocation and the `shmem_barrier_all` on the exit of the call that is performed if the call returns a non-null pointer.

Figure 5.5 illustrates these changes and additions. The dashed red boxes indicate where we applied changes or added functionality to the existing race detection analysis. We adapted this figure from Figure 5.4, which shows the MPI RMA race detection architecture overview in the previous Section 5.1.4.

This figure helps indicate the scope of the changes. However, it is a high-level overview. Thus we will provide insight into the changes or additions in the rest of this section. One of the most significant changes for this and future work made during the development of this OpenSHMEM race detection analysis is creating an XML specification file containing all functions of the OpenSHMEM 1.5 specification.

Before starting this thesis, MUST already contained a specification for OpenSHMEM functions. However, this specification was minimal since it was only used for the generic vector clock module. Therefore this specification only contained functions of the, at the time, current OpenSHMEM 1.4 specification that were relevant for process synchronization. Since a central tenet of OpenSHMEM is the one-sided nature of its communication, this was a small subset of all functions. Specifically, this included the library setup and exit routines, the memory management routines, and the collective routines, such as barriers and broadcasts. Therefore, all remaining functions, which are available in the OpenSHMEM 1.5 specification, were added during the development of this thesis to the existing XML specification. This OpenSHMEM XML specification allows the rest of our and future analyses to access and intercept OpenSHMEM calls.

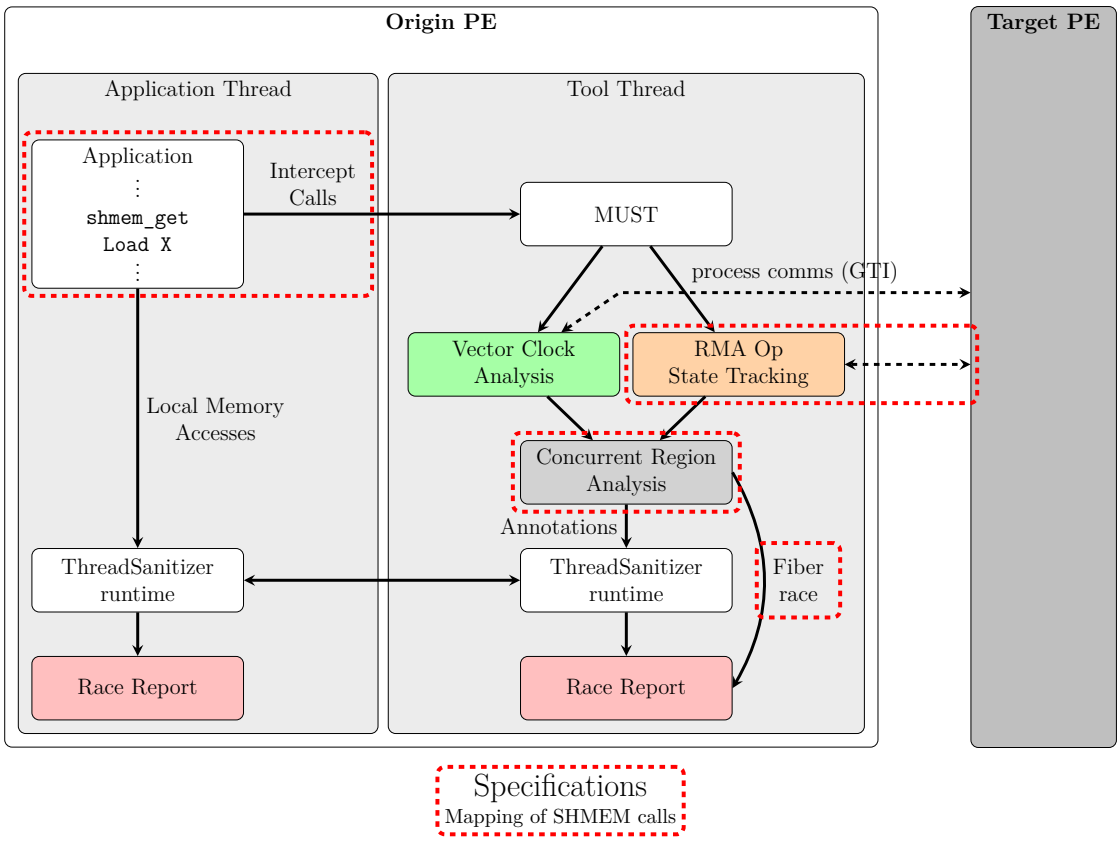


Figure 5.5.: Modifications of the existing MPI RMA race analysis architecture.

For our data race analysis, we implemented a new module called *ShmemTrack*, based on the general design of the RMATrack module. ShmemTrack has similar responsibilities for the OpenSHMEM race analysis as RMATrack does for MPI RMA and replaces RMATrack in our analysis. It provides the analysis functions which directly intercept the OpenSHMEM calls, forwards information to the required remote instances of ShmemTrack, and tracks the application's state. For example, for an RMA communication operation, ShmemTrack passes the information on the accessed memory, atomicity, and other information on the operation to the target. Similarly, when ShmemTrack intercepts an RMA synchronization call, it notifies all involved PEs of this state update. However, all of this functionality depends on the programming model used. While they are similar, there are also significant differences, and thus the functionality of the RMATrack module needed to be newly implemented for the OpenSHMEM race analysis. The components of the ShmemTrack module are shown in Figure 5.6.

Two of the most significant differences in these modules are tracking "public" memory, windows in the case of MPI RMA and OpenSHMEM allocations, and handling RMA synchronization mechanisms. Handling RMA synchronization mechanisms is unique for each and is implemented based on the formal model and API specification. However, the most interesting synchronization routine is `shmem_fence` as MPI RMA

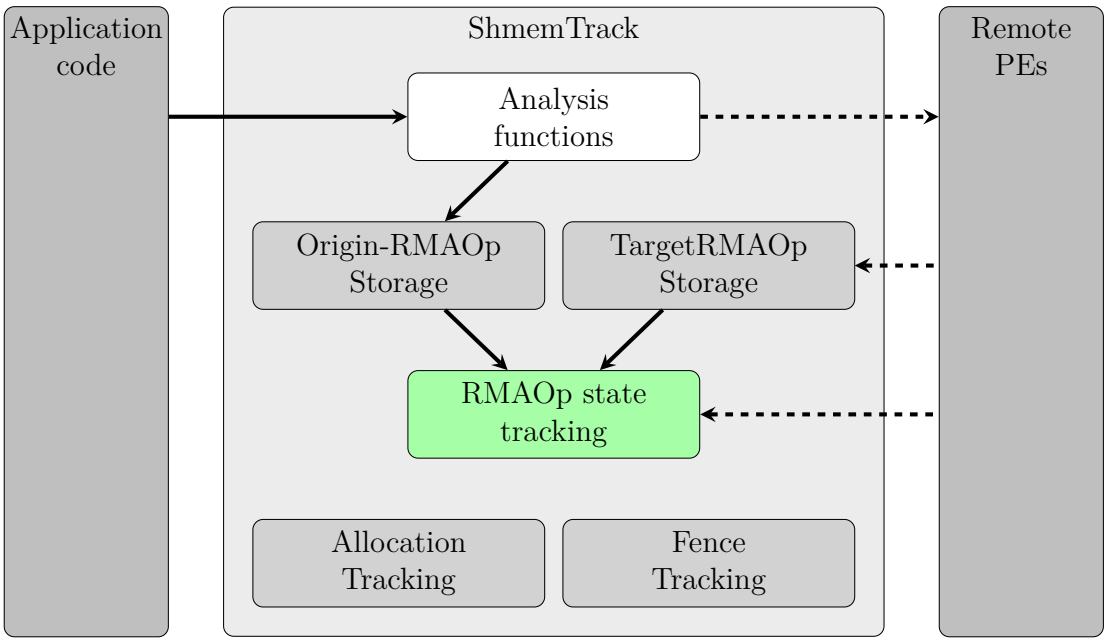


Figure 5.6.: The components contained within the new ShmemTrack module.

provides no direct equivalent. `shmem_fence` has been discussed in detail in Section 2.3.3, but in short, it enforces that all remotely writing operations issued before the `shmem_fence` are delivered before all remotely writing operations with the same origin and target after the fence. With this definition, fences prevent any two remotely writing operations with the same origin and target from conflicting in a data race when at least one fence is issued between them on the origin. Therefore, we must implement a mechanism to prevent a false positive report between those operations.

We implemented this mechanism through a *fence counter*. The fence counter is incremented each time a `shmem_fence` is called, and each time an RMA communication operation is issued, we attach this counter to the corresponding target operation. Our implementation already manually tests for data races coming from the same origin due to the limitations of the fiber annotations described in Section 5.1.3. In this manual data race check, we added a preemptive test that checks whether both operations currently examined operations have the same fence counter. If this is not the case, then a `shmem_fence` must have been issued between these operations, and the check exits early, not reporting a race. Otherwise, the check continues.

The other significant difference of acquiring "public" memory similarly needs an entirely new system. For windows, RMATrack added a window tracking feature in MUST. This system is quite complicated as windows can be of different sizes on different ranks, and window management happens on a communicator basis. However, in OpenSHMEM, memory management of symmetric data objects always occurs globally on the world team, and the allocation sizes cannot differ across PEs. Thus, for a correct allocation, each PE already has all knowledge on the allocation of a

symmetric data object. However, the OpenSHMEM specification does not guarantee that the same pointer is returned on each PE.

We, therefore, implemented a lightweight tracking system that stores the base pointer and size of each allocation locally in the ShmemTrack module. Each allocation is identified by a unique id derived from an *allocation counter* that is incremented on a successful allocation. Since all OpenSHMEM memory management functions occur collectively in the world team, these local counters increment in lockstep. ShmemTrack then passes this allocation id and offset from the base pointer to the remote PE instead of a raw memory location. At the remote PE, ShmemTrack looks up the base pointer of the allocation id, and the actual memory location is calculated.

With these systems and ShmemTrack in place, all necessary information for our race detection mechanism using TSan is available at all PEs. The remaining bulk of the implementation work was spent on developing the intercept functions for OpenSHMEM calls, the processing of their parameters, the interfaces with which ShmemTrack interacts with the preexisting infrastructure, and the mapping of the OpenSHMEM calls to reflect their semantics. Going into detail on this is outside the scope of this thesis. However, the code is provided in [19], and the mapping of the OpenSHMEM calls mainly derives from the formal model of OpenSHMEM we developed in Chapter 4.

6. Classification quality evaluation

The architecture of our prototype OpenSHMEM data race detection tool has been presented in Chapter 5. In this chapter, the data race test set we developed in Chapter 3 is used to test the prototype and evaluate its performance. As discussed in Chapter 3, our test set is limited to minimal test cases developed during this thesis's writing. As this does not include any codes or benchmarks useful for comparing performances between execution, this chapter will only evaluate the detection accuracy of our prototype.

As accurate data race detection is known to be NP-hard [34], data race detection tools used in practice tolerate some incorrect results. As our tool builds on pre-existing tools which tolerate such results, this will also be the case for the prototype in this thesis.

6.1. Testing setup

Before presenting the results, the testing setup needs to be established to provide context and repeatability to the test results. The prototype was developed and tested in a container run through the Podman [36] container manager. The container is based on the *Ubuntu 22.04.1 LTS* image from the Docker repositories. Furthermore, our prototype and the test cases have been compiled and instrumented using the LLVM-12 toolchain provided through Ubuntu's packages. Similarly, the image installed the MPI library required for GTI through the Ubuntu package manager, for which we chose MPICH version 4.0. The last missing central component from this setup is an OpenSHMEM library. For this, we had a few options. However, at the start of this thesis, only Sandia OpenSHMEM [6] supported the full OpenSHMEM 1.5 specification [8], which is the specification used in this thesis. Thus, we chose Sandia OpenSHMEM as the OpenSHMEM implementation.

However, this OpenSHMEM implementation is unavailable through the Ubuntu package manager and must be built and installed manually. In order to prevent incompatibilities, all software is compiled using the LLVM/Clang 12 compiler. An image of our environment can be built using the docker build file in the publication artifacts for this thesis [19]. Lastly, all tests are integrated into the MUST repository under `tests/OneSidedChecks/OpenSHMEM`. The repository, the tests it contains, and other thesis artifacts are available through RWTH Aachen publications service [19]. The tests are run using the build system or directly through the LLVM test infrastructure (lit). The test results used in the remainder of this thesis were generated this way.

6.2. Classification quality study

In a 2020 paper [52], Verma et al. follow up and improve upon DataRaceBench. Alongside general improvements to the test suite, they used a set of statistical metrics to measure the capabilities of different data race detection tools.

The data on which these metrics are calculated is the number of tests that were correctly and incorrectly identified as containing or not containing a data race by the tool. These are the True Positives (TP), the cases where the tool correctly identifies a race in a test, and the False Positives (FP), where the tool incorrectly identifies a race or in other words produces a false alert. Similarly, these are the False Negatives (FN), where the tool does not identify a race when the test contains one, and True Negatives (TN), where the tool correclty does not report a race in a test that does not contain a race.

The metrics given in Table 6.1 are widely used statistical measures in binary classification tests used to measure the performance of different tools, such as image recognition software and, in our case, correctness tools. They are helpful in comparing the performance of correctness tools without the need to directly compare their behavior on individual test cases. For example, if one tool has a higher *precision* than another, the former reports fewer false positives on a given test set. In comparison, *Recall* indicates the rate of correctly identified from all reported positive results.

Tool Result	Ground Truth		Recall	Specificity	Precision	Accuracy	F1 Score
	True	False					
True	TP	FP	TP / (TP + FN)	TN / (TN + FP)	TP / (TP + FP)	(TP+TN) / (TP + FP + TN+ FN)	2 * (P * R) / (P + R)
False	FN	TN					

Table 6.1.: Definition of metrics (Recall, Specificity, Precision, Accuracy and F1 Score) [52, TABLE II]

As discussed in Section 3.3, there was no previous work providing benchmarks or tests and, therefore, no pre-existing source to generate this data for the OpenSHMEM race detection prototype developed in this thesis. Thus, we had to develop test cases to test our prototype against them. In this thesis, we developed a total of 35 test cases which were further described in Section 3.3. However, in short, these tests are intended to be minimal examples that cover most of the functionality provided by OpenSHMEM and the combinations in which it can be used. Of these tests, 21 contained data races in which OpenSHMEM functionality is used, while the remaining 14 tests contained no data race. Each test case follows the same structure, isolating the actual race from the rest of the boilerplate code.

Running this test suite against our prototype implementation results in the raw output in Figure A.1. During this run, six tests are skipped as they are marked unsupported since these features are not yet implemented in our prototype:

- `shmem_ctx_race.c`
- `shmem_ctx_safe.c`
- `shmem_lock_race.c`

- `shmem_lock_safe.c`
- `shmem_ptr_race.c`
- `shmem_ptr_safe.c`

With six unsupported tests of a total of 35 tests, we have a test support rate (TSR) of 82.86 %. The remaining 29 test cases, 18 of which contain a race, all only contain supported features that we can test against our prototype. The results of this measurement run and the consequent metrics are given in Table 6.2:

Tool Result	Ground Truth		Recall	Specificity	Precision	Accuracy	F1 Score
	True	False					
True	14	0	0.7777	1.0	1.0	0.8621	0.875
False	4	11					

Table 6.2.: Classification quality of the OpenSHMEM data race detection prototype.

These results are promising for our prototype implementation. Recall measures the proportion of races that were correctly identified. It indicates how well our tool identifies races that exist in the code. For our prototype, recall is not an optimal score, as we had 4 cases where our prototype did not correctly identify races. These false negative results will be discussed in the following Section 6.3.

In the same vein, specificity measures how well our prototype correctly identifies test cases where no races are present. For our prototype, specificity is an optimal score since we had no false positive results, which also results in optimal precision. As false positives in data race detection tools are often signs of a tool not understanding a synchronization in a test, this indicates that our implementation can correctly understand all supported synchronization mechanisms with some limitations. For some test cases, in particular, annotations in the application are necessary to provide the data race detection with the required information, as this information cannot be gathered simply by intercepting function calls. For example, for a `shmem_wait_until` operation, our prototype needs information on the second operation accessing the value on which `shmem_wait_until` is waiting. However, the specifics of these limitations will be discussed in the following Section 6.3.

The fact that our tool reports no false positives (with external annotations) is beneficial for developers of applications as they can be sure that our tool will not report a race that does not exist, leading them to a search for a non-existent bug. This lack of false positives allows the developer to apply the prototype in an iterative fashion where initially, the tool can be applied without any annotations, and if it reports a race, the developer can check whether a race truly takes place. Our tool even reports the stack trace for both accesses through TSan, and additional reports on the PE from which the stack trace is coming. An example output is given in Figure 6.1. This feature further simplifies this workflow and allows the developer to quickly find the reported race and check whether it is a race or if an annotation is missing. After which, the developer can apply the correct annotations if necessary,

```
==================
WARNING: ThreadSanitizer: data race (pid=584)
  Write of size 1 at 0x7fa080002250 by thread T17:
    #0 shmem_int_put /home/user/build/must/openshmem-new/externals/
   GTI/externals/PnMPI/src/pnmpi/shm_wrapper_c.c:88290 (libpnmpi.so
   +0x39c430)
    #1 <null> <null> (0xffffffffffffffff)
    #2 <null> <null> (0x000100000000)
    #3 <null> <null> (0x000100000003)
    #4 main /home/user/sourcecode/general_examples/
   shmem_default_race.c:34:9 (a.out+0x4b9c2c)

  Previous read of size 4 at 0x7fa080002250 by main thread:
    #0 main /home/user/sourcecode/general_examples/
   shmem_default_race.c:41:51 (a.out+0x4b9c48)

  Thread T17 'rank 0' (tid=0, running) created by thread T3 at:
    #0 must::TSan::createFiber(unsigned int) /home/user/MUST/
   modules/TSan/TSan.cpp:183:50 (libtSan.so+0x7d4c6)

SUMMARY: ThreadSanitizer: data race /home/user/build/must/openshmem
   -new/externals/GTI/externals/PnMPI/src/pnmpi/shm_wrapper_c.c
   :88290 in shmem_int_put
==================
```

Figure 6.1.: Output for `shmem_default_race.c`.

which will, with certainty, remove the race report if the synchronization and its annotations are correct.

In total, our accuracy, or the ratio of correct results out of all results, is 86.21 %. This accuracy makes this prototype very useable in practice, as data race detectors for other programming models [52] have similar ranges in accuracy. Such accuracy allows developers to find the majority of all data races using this prototype, which will likely significantly reduce the development time of applications [54].

6.3. Test case analysis

After applying the test suite to our prototype, the prototype failed to detect data races in four tests. These tests are listed below:

- `shmem_atomic_type_mixed_race.c`
- `shmem_atomic_local_mixed_race.c`
- `shmem_put_signal_race.c`
- `shmem_put_nbi_arr_race.c`

In the first test case, `shmem_atomic_type_mixed_race.c`, the failure was expected due to the design of our tool. As we use TSan as our central race detection component,

by annotating memory accesses through its interface, we are also limited by it. As discussed in Section 2.2.2, OpenSHMEM only guarantees atomicity for its AMOs under certain conditions. One of these conditions is matching the data type of the two conflicting AMOs. However, TSan is designed to be a cross-platform shared-memory data race detection tool and thus has no notion of data type, only the size of accesses where a mismatched size does not necessarily result in a race. We can, therefore, not detect data races of two conflicting AMOs using mismatched datatypes using TSan.

The next two failed tests are the `shmem_atomic_local_mixed_race.c` test and the `shmem_put_signal_race.c` test, in both of these tests OpenSHMEM atomic operations, `shmem_int_atomic_set` and the signal operation of `shmem_int_put_signal` respectively, conflict with accesses that are not issued through an OpenSHMEM call. Within our prototype, we annotate the target operations of OpenSHMEM AMOs using the TSan atomic load/store interface. However, the local access is non-atomic. Specifically, the variable is of type static C/C++ `int`. As this type is not guaranteed to be atomic, annotating the local access OpenSHMEM operation as non-atomic should result in a data race, however, this is not the case. At the time of writing, there is no solution to this problem, as it seems to be the intended behavior of TSan. These two tests and the previous test display a pattern that our prototype fails in test cases where the data race happens due to circumstances we described with the label *false atomicity* in Section 3.2. As described, all this behavior is due to the limitations of TSan, as it was only designed to be a shared-memory data race detector.

In contrast, the last of the four failed tests is `shmem_put_nbi_arr_race.c` and fails for a different reason. In this test, we use `shmem_int_put_nbi` to transfer an array of integers between PEs, and the source array is modified on the origin after the non-blocking put resulting in a race. This origin operation is correctly annotated in TSan, as they are in all other test cases that work correctly and contain similar local buffer races. However, no race is detected by the TSan runtime, and at the time of writing, the cause for this behavior is unknown.

While some test cases work out of the box without any modifications, others without data races need annotations, as described in the previous Section 6.2. All of these tests, use some resource-based synchronization mechanisms, such as signaling or point-to-point synchronization. As previously mentioned, these annotations are required since these synchronization mechanisms require the context of the application code, which is unavailable by simply intercepting their function calls. Thus, a developer needs to annotate the application and provide the required information for the OpenSHMEM data race detection analysis. Within the test suite, this is achieved by including the `GTI_Annotations.h` header, which gives access to annotation functions such as `GTI_AnnotateTick()` that increments the vector clock on the calling PE. Without these annotations, the tests `shmem_wait_until_safe.c` and `shmem_put_signal_polling_safe.c` would fail by reporting a false positive. However, in practice, this is an acceptable tradeoff for a tool as it significantly decreases complexity on the side of the tool and allows for easy use of the tool, while it is only a little effort for the developer.

The last deficiency of the prototype developed in this thesis is the unsupported features. The three unsupported features are communication contexts, distributed locks, and `shmem_ptr`. Supporting communication contexts and distributed locks should be a matter of more implementation work, as the essential components required for their support are already present. In short, communication contexts are always associated with a team and present a method to apply memory ordering routines selectively. As the prototype already supports teams, it should be possible to track communication contexts by intercepting their creation routines and associating each context with a unique id. Each time an operation is issued, the id for the given context could be attached to the target operation. Based on this id, the operations could be selectively completed at the target, similar to how support for `shmem_fence` was achieved.

Supporting distributed locks should also prove achievable, as the vector clock module in MUST already supports mutual exclusion in terms of locks. However, a call to `shmem_clear_lock` also performs a `shmem_quiet` operation on the default context before releasing the lock. Thus, all operations issued on the default context are completed before the lock is released. This behavior would need to be modeled in the prototype, and additional development time would need to be spent integrating the distributed locks into the existing vector clock infrastructure, which was missed due to time constraints. Lastly, `shmem_ptr` might prove challenging to implement. In short, `shmem_ptr` allows accessing remote memory directly through a pointer, as expected from a standard C/C++ pointer. These accesses can not be tracked through P^{N}SHMEM as they are not issued through any OpenSHMEM call. Thus, we do not support `shmem_ptr`. However, the specification does not require this feature to be constantly available, and a developer needs to perform runtime checks to test its availability. Due to this, in many implementations, `shmem_ptr` is not supported at all or only through special features by system vendors, and we do not expect this feature to be widely used.

7. Conclusion

OpenSHMEM as a programming model, is focused on one-sided communication. The one-sided nature of communication allows overlap with computation in an application, making OpenSHMEM ideal for unstructured, small-to-medium-sized data communication patterns [8, Ch. 2]. However, one-sided communication also increases the probability of programming mistakes as the interaction between processes where such errors might become apparent is limited. Data races are particularly inconspicuous for developers of OpenSHMEM applications as public memory is freely accessible for regular application code.

To this end, this thesis provides several elements that are beneficial to future OpenSHMEM efforts, particularly the detection of data races. Examining the semantics of the OpenSHMEM specification revealed many situations where OpenSHMEM and language-native features can result in a data race. These races often shared common patterns and properties, such as violations of the conditions required for the atomicity of atomic OpenSHMEM operations. In order to allow reasoning about these data races without the explicit use of their source code, a labeling system for the data race examples was introduced in this thesis.

At the time of writing, no previous work provided existing codes or data race benchmarks that could be used to evaluate the prototype developed during this thesis. Thus, in this thesis, the data race example set was expanded and developed into 35 test cases. These test cases are minimal examples only intended to test the classification accuracy of data race detectors for OpenSHMEM. However, they are independent of our tool and can be used to evaluate any other future tools.

Further, in previous work [13] and during the examination of the OpenSHMEM specification, significant similarities between OpenSHMEM and MPI RMA became apparent. Based on this observation, the semi-formal model by Hoefler et al. [17] was adapted to the semantics of OpenSHMEM in this thesis. This model examines an individual execution of a program and represents the individual program statements and their effects using actions that are ordered using the happens-before and consistency relations.

This adaption allows determining the concurrent regions for OpenSHMEM operations using the happens-before and consistency relation. Concurrent regions represent the interval in logical time where an action can occur. If the concurrent regions of two conflicting memory accesses overlap, they are in a data race. Therefore tracking the concurrent regions of operations allows for detecting data races. This data race detection approach was used to implement a data race detection tool for MPI RMA [45] based on this described data race detection approach and the model by Hoefler et al. This thesis combines the adaptation of the semi-formal model

and the existing data race detection implementation for MPI RMA by using it as a foundation for implementing a data race detection prototype in OpenSHMEM.

The prototype is capable of understanding most of OpenSHMEM's features. However, some features, such as distributed locks or communication contexts, are unsupported. Evaluating this prototype with the data race test set developed earlier in this thesis shows promising results with an accuracy of 0.86. Notably, the prototype reports no false positive results in the supported tests.

7.1. Future work

The results in this thesis are promising for OpenSHMEM tool support. The prototype developed in this thesis shows that it is possible to detect a majority of data races in OpenSHMEM codes and that the semantics of OpenSHMEM can be modeled to develop an understanding of OpenSHMEM routines and their interaction with the rest of the application code. However, the prototype and model are not fully featured yet. The prototype lacks support for some features available in the specification and would need more work. This is also the case for the model as it was restricted to model only a subset of the specification due to time and complexity restraints. Additionally, the problem with TSan as the central race detector for our prototype and its problem detecting races in tests labeled with "false atomicity" requires more research.

Similarly, the test cases developed in this thesis are an initial set of tests helpful in evaluating data race detection tools for OpenSHMEM. However, their number is also limited. While the semantics-driven search for data races covered most of the specification, the test set still needs expansion. More work is needed on the test set, with tests using a greater variety of OpenSHMEM calls and tests helpful in gauging the overhead the prototype imposes.

A. Data Race Classification

```
user@sandia:~/build/must/openshmem-new$ lit -s -j 1 tests/OneSidedChecks/OpenSHMEM/
FAIL: MUST :: OneSidedChecks/OpenSHMEM/shmem_put_nbi_arr_race.c (29 of 35)
UNRESOLVED: MUST :: OneSidedChecks/OpenSHMEM/shmem_ctx_safe.c (30 of 35)
UNRESOLVED: MUST :: OneSidedChecks/OpenSHMEM/shmem_lock_safe.c (31 of 35)
UNRESOLVED: MUST :: OneSidedChecks/OpenSHMEM/shmem_ptr_safe.c (32 of 35)
UNRESOLVED: MUST :: OneSidedChecks/OpenSHMEM/shmem_ctx_race.c (33 of 35)
UNRESOLVED: MUST :: OneSidedChecks/OpenSHMEM/shmem_lock_race.c (34 of 35)
UNRESOLVED: MUST :: OneSidedChecks/OpenSHMEM/shmem_ptr_race.c (35 of 35)
********************
Unresolved Tests (6):
  MUST :: OneSidedChecks/OpenSHMEM/shmem_ctx_race.c
  MUST :: OneSidedChecks/OpenSHMEM/shmem_ctx_safe.c
  MUST :: OneSidedChecks/OpenSHMEM/shmem_lock_race.c
  MUST :: OneSidedChecks/OpenSHMEM/shmem_lock_safe.c
  MUST :: OneSidedChecks/OpenSHMEM/shmem_ptr_race.c
  MUST :: OneSidedChecks/OpenSHMEM/shmem_ptr_safe.c

********************
Failed Tests (1):
  MUST :: OneSidedChecks/OpenSHMEM/shmem_put_nbi_arr_race.c


Testing Time: 127.57s
  Passed           : 25
  Expectedly Failed:  3
  Unresolved       :  6
  Failed           :  1
```

Figure A.1.: Raw output of running the test suite against The OpenSHMEM race detection prototype developed in this thesis.

Race name	Is race?	Race property labels	Synchronization mechanisms labels
atomic local mixed	yes	remote race, false atomicity	atomics
atomic nbi local	yes	local buffer race, false atomicity	atomics
atomic nbi	no	safe	atomics, memory-order
atomic	no	safe	atomics
atomic type mixed	yes	remote race, false atomicity	atomics
collective reduce	yes	remote race	process-sync
collective reduce	no	safe	process-sync, memory-order
collective team reduce	yes	remote race	process-sync
collective team reduce	no	safe	process-sync, memory-order
ctx	yes	remote race	memory-order
ctx	no	safe	memory-order
default	yes	remote race	
fence	yes	remote race, inconspicuous reordering	memory-order
fence	no	safe	memory-order
lock	yes	remote race, unreliable manifestation	resource-based sync
lock	no	safe	resource-based sync
ptr	yes	remote race	
ptr	no	safe	process-sync, memory-order
put get 2proc	yes	remote race, local buffer race	
put get 3proc	yes	remote race	
put load	yes	remote race	
put nbi arr	yes	local buffer race	
put nbi local	yes	local buffer race	
put nbi remote	yes	remote race, local buffer race	
put nbi	no	safe	process-sync, memory-order
put signal polling	no	safe	resource-based sync
put signal	yes	remote race	resource-based sync
put store local	yes	remote race	
quiet nbi get	no	safe	process-sync, memory-order
quiet	yes	remote race, inconspicuous reordering	memory-order
quiet	no	safe	memory-order
realloc	no	safe	process-sync, memory-order
remote put race	yes	remote race	
wait until	yes	remote race, flase atomicity	atomics, resource-based sync
wait until	no	safe	atomics, memory-order, resource-based sync

Table A.1.: Labeling of all test cases.

Bibliography

[1] Advanced Micro Devices, Inc. ROCm SHMEM. `https://github.com/ROCm-Developer-Tools/ROC_SHMEM`. Accessed: 2022-11-30.

[2] T. C. Aitkaci, M. Sergent, E. Saillard, D. Barthou, and G. Papauré. Dynamic Data Race Detection for MPI-RMA Programs. In *EuroMPI 2021-European MPI Users's Group Meeting*, 2021.

[3] G. Almasi. *PGAS (Partitioned Global Address Space) Languages*, pages 1539–1545. Springer US, Boston, MA, 2011.

[4] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller. ARCHER: Effectively spotting data races in large openmp applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 53–62, 2016.

[5] M. A. S. Bari, U. Arora, V. Hegde, T. Curtis, and B. Chapman. OpenSHMEM Checker - A Clang based static checker for OpenSHMEM. In *2021 20th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 41–48. IEEE, 2021.

[6] B. W. Barrett, S. Smith, J. Dinan, K. Seager, and R. E. Grant. Sandia openshmem, version 00, 3 2016.

[7] H.-J. Boehm and S. V. Adve. Foundations of the C+++ concurrency memory model. *SIGPLAN Not.*, 43(6):6878, jun 2008.

[8] O. Committee. *OpenSHMEM Application Programming Interface Version 1.5.*

[9] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[10] J. FIDGE. Timestamps in message-passing systems that preserve the partial ordering. *Proc. 11th Australian Comput. Science Conf.*, pages 56–66, 1987.

[11] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling highly-scalable remote memory access programming with mpi-3 one sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Google Inc. Thread sanitizer algorithm. `https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm`. Accessed: 2023-03-16.

[13] J. R. Hammond, S. Ghosh, and B. M. Chapman. Implementing openshmem using mpi-3 one-sided communication. In *Workshop on OpenSHMEM and Related Technologies*, pages 44–58. Springer, 2014.

[14] O. Hernandez, S. Poole, J. Kuehn, S. Jana, S. Pophale, and B. Chapman. The OpenSHMEM analyzer. In *PGAS12, 2012*, 2012.

[15] 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 *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1364–1375, 2012.

[16] T. Hilbrich, M. Schulz, B. R. d. Supinski, and M. S. Müller. MUST: a scalable approach to runtime error detection in MPI programs. In *Tools for high performance computing 2009*, pages 53–66. Springer, 2010.

[17] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood. Remote memory access programming in mpi-3. *ACM Transactions on Parallel Computing (TOPC)*, 2(2):1–26, 2015.

[18] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, fourth edition, June 2018.

[19] S. Klotz. Thesis artifacts. `https://publications.rwth-aachen.de/record/953855`.

[20] B. Krammer, K. Bidmon, M. Müller, and M. Resch. MARMOT: An mpi analysis and checking tool. In G. Joubert, W. Nagel, F. Peters, and W. Walter, editors, *Parallel Computing*, volume 13 of *Advances in Parallel Computing*, pages 493–500. North-Holland, 2004.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558565, jul 1978.

[22] M. Laurent, E. Saillard, and M. Quinson. The MPI Bugs Initiative: a Framework for MPI Verification Tools Evaluation. In *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 1–9, 2021.

[23] Lawrence Livermore National Laboratory. DataRaceBench 1.4.0 released. `https://software.llnl.gov/news/2021/10/22/dataracebench-14.0/`, Oct. 2021. Accessed: 2022-01-11.

[24] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin. DataRaceBench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[25] C. Liao, P.-H. Lin, M. Schordan, and I. Karlin. A semantics-driven approach to improving dataracebenchs OpenMP standard coverage. In *International Workshop on OpenMP*, pages 189–202. Springer, 2018.

[26] LLVM Developer Group. GCC, the GNU compiler collection. `https://gcc.gnu.org/`. Accessed: 2023-03-16.

[27] LLVM Developer Group. The LLVM compiler infrastructure. `https://llvm.org/`. Accessed: 2023-03-16.

[28] LLVM Developer Group. Threadsanitizer implementation in the LLVM project. `https://github.com/llvm/llvm-project/tree/main/compiler-rt/lib/tsan`. Accessed: 2023-02-23.

[29] LLVM Developer Group. Fiber support for thread sanitizer. `https://reviews.llvm.org/D54889`, Nov. 2018. Accessed: 2023-02-23.

[30] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. *ACM SIGPLAN Notices*, 40(1):378–391, 2005.

[31] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.0*, July 1997.

[32] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*, Sept. 2012.

[33] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[34] R. H. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[35] Nvidia Corporation. NVSHMEM. `https://developer.nvidia.com/nvshmem`. Accessed: 2022-11-30.

[36] podman. `https://podman.io/`. Accessed: 2023-03-16.

[37] S. Pophale, O. Hernandez, S. Poole, and B. M. Chapman. Extending the OpenSHMEM Analyzer to perform synchronization and multi-valued analysis. In S. Poole, O. Hernandez, and P. Shamis, editors, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools*, pages 134–148, Cham, 2014. Springer International Publishing.

[38] S. S. Pophale, A. Curtis, B. Chapman, and S. Poole. Poster: Validation and verification suite for OpenSHMEM. In *Proceedings of the Seventh Conference on Partitioned Global Address Space Programming Model (PGAS 2013)*, volume 257, page 258, 2013.

[39] J. Protze. *Modular techniques and interfaces for data race detection in multi-paradigm parallel programming.* Dissertation, RWTH Aachen University, Aachen, 2021. Druckausgabe: 2021. - Auch veröffentlicht auf dem Publikationsserver der RWTH Aachen University; Dissertation, RWTH Aachen University, 2021.

[40] A. Raza. A review of race detection mechanisms. In D. Grigoriev, J. Harrison, and E. A. Hirsch, editors, *Computer Science – Theory and Applications*, pages 534–543, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[41] M. Schulz and B. R. De Supinski. A flexible and dynamic infrastructure for mpi tool interoperability. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 193–202, 2006.

[42] M. Schulz and B. R. de Supinski. PNMPI Tools: A whole lot greater than the sum of their parts. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, New York, NY, USA, 2007. Association for Computing Machinery.

[43] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing*, 7(3):149–174, 1994.

[44] S. Schwitanski. On-the-Fly Data Race Detection in MPI One-Sided Communication. Masterarbeit, RWTH Aachen University, Aachen, 2017. Masterarbeit, RWTH Aachen University, 2017.

[45] S. Schwitanski, J. Jenke, F. Tomski, C. Terboven, and M. S. Müller. On-the-Fly Data Race Detection for MPI RMA Programs with MUST. In *Proceedings of Correctness 2022: 6th International Workshop on Software Correctness for HPC Applications, Held in conjunction with SC 2022: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 27–36, Piscataway, NJ, Nov 2022. 6.International Workshop on Software Correctness for HPC Applications, Dallas, TX (USA), 13 Nov 2022 - 18 Nov 2022, IEEE. Date Added to IEEE Xplore: 31 January 2023. - Zweitveröffentlicht auf dem Publikationsserver der RWTH Aachen University 2023.

[46] S. Schwitanski, F. Tomski, J. Protze, C. Terboven, and M. S. Müller. An on-the-fly method to exchange vector clocks in distributed-memory programs. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 530–540. IEEE, 2022.

[47] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[48] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[49] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with llvm compiler. In *International Conference on Runtime Verification*, pages 110–114. Springer, 2011.

[50] K. Serebryany and D. Vyukov. Finding races and memory errors with compiler instrumentation. `https://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=kcc.pdf`. Accessed: 2023-03-16.

[51] F. Tomski. Developing an on-the-fly vector clock exchange for distributed memory systems using the generic tool infrastructure. Masterarbeit, RWTH Aachen University, Aachen, 2021. Masterarbeit, RWTH Aachen University, 2021.

[52] G. Verma, Y. Shi, C. Liao, B. Chapman, and Y. Yan. Enhancing dataracebench for evaluating data race detection tools. In *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 20–30, 2020.

[53] J. Vetter and B. de Supinski. Dynamic software testing of mpi applications with Umpire. In *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 51–51, 2000.

[54] S. Wienke, J. Miller, M. Schulz, and M. S. Müller. Development effort estimation in hpc. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 107–118, Nov 2016.