Surviving Errors with OpenSHMEM

Aurelien Bouteiller[1(✉)], George Bosilca[1], and Manjunath Gorentla Venkata[2]

[1] Innovative Computing Laboratory, University of Tennessee, Knoxville, USA
{bouteill,bosilca}@icl.utk.edu
[2] Oak Ridge National Laboratory, Oak Ridge, USA
manjugv@ornl.gov

Abstract. Unexpected error conditions stem from a variety of underlying causes, including resource exhaustion, network failures, hardware failures, or program errors. As the scale of HPC systems continues to grow, so does the probability of encountering a condition that causes a failure; meanwhile, error recovery and run-through failure management are becoming mature, and interoperable HPC programming paradigms are beginning to feature advanced error management. As a result from these developments, it becomes increasingly desirable to gracefully handle error conditions in OpenSHMEM. In this paper, we present the design and rationale behind an extension of the OpenSHMEM API that can (1) notify user code of unexpected erroneous conditions, (2) permit customized user response to errors without incurring overhead on an error-free execution path, (3) propagate the occurence of an error condition to all Processing Elements, and (4) consistently close the erroneous epoch in order to resume the application.

1 Introduction

OpenSHMEM [21] is an emerging partitioned global address space (PGAS) specification that provides interfaces for one-sided and collective communication, synchronization, and atomic operations. The one-sided communication operations do not require the active participation of the target process when receiving or exposing data, freeing the target process to work on other tasks while the data transfer is ongoing. It also supports some collective communication patterns such as synchronizations, broadcast, collection, and reduction operations. In addition OpenSHMEM provides interfaces for a variety of atomic operations including

M.G. Venkata—This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

M. Gorentla Venkata et al. (Eds.): OpenSHMEM 2016, LNCS 10007, pp. 66–81, 2016.
DOI: 10.1007/978-3-319-50995-2_5

both 32-bit and 64-bit operations. Overall, it provides a rich set of interfaces for implementing parallel scientific applications, and OpenSHMEM implementations are expected to perform well on modern high performance computing (HPC) systems. This expectation stems from the design philosophy of OpenSHMEM, which focus on providing a lightweight and high performing minimalistic set of operations, and a close match between the OpenSHMEM semantic and hardware-supported native operations. This tight integration between the hardware and the programming paradigm is expected to result in close to optimal latency and bandwidth in synthetic benchmarks, meanwhile preserving simple and powerful end-user semantics.

Despite this rich feature set, the OpenSHMEM specification has lacked error management and failure mitigation primitives. However, the complexity of High Perfomance Computing systems keeps increasing steadily along multiple axes. On one axis, heterogeneous computing, with accelerators, different instruction sets, and possibly multiple interoperable programming paradigms are becoming pervasive [13]. This proliferation of software levels within the same application increases the probability of hitting unforeseen interactions between the runtime libraries, leading, in the worse case, to more programming errors from the more numerous code paths, or to imperfect resource sharing between levels, hitherto more occurrences of runtime resource exhaustion errors. Along another axis, HPC is moving further toward massive parallelism, harnessing millions of processing cores, in commonly tens of thousand of nodes. As the number of components comprising HPC systems increases, probabilistic amplification entails that failures (*i.e.,* a system malfunction) are becoming common events in the lifecycle of an application. Currently deployed petascale machines experience approximately one crash failure every 10 h [22], a situation which is expected to worsen with the introduction of exascale systems in the near future [1,7,14]. Although some faillures may not be immediately visible (especially the so called *silent errors* that corrupt the application dataset without interrupting the computation), in many cases, failures (including a large number of memory corruptions) do manifest detectable behavior, either in the form of a process crash, a network disconnect, or as a memory corruption that can't be corrected by ECC.

As these failure vectors become more common, most HPC programming interfaces are being enriched to provide meaningful error reporting and mitigation strategies. For example, the Message Passing Interface (MPI) has long provided error reporting capabilities, and further semantics to tolerate process failures are under consideration [5]. In this paper we present a set of extensions to the OpenSHMEM specification that will enable capturing errors resulting from various unexpected runtime conditions, stabilize the state of the application—and thereby open the possibility for recovering from the condition, and possibly interoperate with another error managing middleware. Due to its one-sided nature, and the form in which synchronization are expressed, OpenSHMEM poses a specific set of constraints for resolving the global state generated by the occurence of unexpected errors at some PEs, which in turns calls for an original approach. The rest of this paper is organized as follows: Sect. 2 presents a

succint view of the type of failures we address and the general benefits expected from their handling; Sect. 3 describes the limitations on error reporting scope and uniformity accross process to preserve latency sensitive operations' performance; Sect. 4 presents the OpenSHMEM API to capture errors; Sect. 5 discusses the need for, and the mean for, the error propagation mechanism; Sect. 6 presents the construct for stabilizing the post-error situation, and resuming communicating; Sect. 7 presents related work on fault tolerant communication libraries, and we conclude in Sect. 8.

2 Background

Error masking and automatic failure recovery are valuable properties in system designs. Indeed, they relieve end-users from the duress of managing erroneous cases, and abstract the system as a stable platform. Aside from component hardware technologies, like packet retransmission in network interfaces, and ECC memory, the main vessel for sustaining the abstraction of a stable platform has been Coordinated Checkpoint/Restart (CR), either at the application or at the system level. One of its strong features is that it can be implemented without the communication library providing a meaningful support for fault tolerance, or even error reporting. In exchange, the recovery strategy involves a coarse grain full restart of the application in the previously saved global state. However, models and analysis [7,14] indicate that the status-quo is not sustainable, and either CR must drastically improve (for example by deploying in-place checkpointing [3,19]), or alternative recovery strategies must be considered. The variety of prospective techniques is wide, and notably includes checkpoint-restart variations based on uncoordinated rollback recovery [9], replication [14], or algorithm based fault tolerance—where mathematical properties are leveraged to avoid checkpoints [12]. A common feature required by most of these advanced failure recovery strategies is that, unlike historical rollback recovery, the application continues to operate in-line and in-place, possibly only demanding the replacement of a limited number of processors. Furthermore, considering the general spectrum of causes that can trigger an error, not all errors are indicative of a catastrophic, or at least severe enough failure, as to justify a full, expensive restart of the platform. The first step to enable an alternative management of errors, or simply to enable scalable checkpointing, is to introduce a mean to report errors from the application's communication support environment.

Failures can be classified into four broad categories of increasing severity. Note that these failure classes do not necessarily map directly to a particular type of ailment; for example, both a memory corruption and an incorrect program can result in a crash failure, or, depending on runtime conditions, both may also produce a silent error, arguably a more severe outcome. In this section we discuss these failure classes' details, and how the OpenSHMEM error reporting system can help their management.

Resource Exhaustion: The first class, which generally is the easiest to circumvent, represents resource exhaustion errors, and other correctable conditions arising

from temporary or maleable overload of capacities. In the general OpenSHMEM philosophy, these errors should generally be handled internally by the library itself, and never propagate to the end-user. However, in some cases, the automatic, internal circumvention of an error is not possible. A program that tries to allocate a large amount of symmetric memory is a simple example. On some architectures, the memory that can be exposed for direct one-sided operations is smaller than the general memory capacity. Should a program require more than the available capacity, a potential corrective action could be to move the least used dataset from a symmetric memory segment to some non-registered memory. However, the OpenSHMEM implementation does not hold enough information about the intent of the application to safely undergo such an action: other Processing Elements (PEs, which is the name for an OpenSHMEM process) may initiate one-sided operations targeting these segments, and it would be unsafe to displace them. As a consequence, an implementation may be forced to report that the symmetric memory is exhausted, and delegate remediation actions to the user's program. These actions could range from operating with a smaller dataset when the algorithm is amenable to such an outcome, or moving some least used symmetric memory to non-symmetric memory explicitly, or continuing with an alternative interoperable communication library to complete the program successfully, albeit with reduced performance.

Crash Failures: The second class captures simple crash failures. A crash failure is characterized by the fact that some PEs stop being responsive definitively, that is, they no longer emit messages. Aside from obvious power supply failures, multiple vectors (including failures of network cables, bit-flips that raise signals, etc.) can ultimately manifest as a PE exhibiting a crash failure. Crash failure detection in distributed systems is a well studied domain [10], with practical solutions [6], outside the scope of the present work. One may note, however, that in a distributed system, surveillance of every process by every process can generate a significant amount of noise, which in turn cause a significant performance degradation [20]. Meanwhile, performing periodic failure resilient consensus to agree upon a set of failed processes is expensive. As a consequence, in practice, failures are detected opportunistically, and a PE may know, at any instant, of only a subset of the full set of failed PEs (as could be observed from an omniscient observer). A desired property with respect to an OpenSHMEM implementation is that it should be free of deadlocks, even when some PE fails, which means that OpenSHMEM operations trigger an appropriate error when PE failures are detected.

Network Failures: The third class is network failures and intermittent failures. These failures manifest when network links and processors are slow, when link failures result in partial disconnection of the network (that is, a PE may appear non-responsive to some neighbors, but responsive to others), or when network messages are lost. Traditionally, message losses and retransmission are easily managed internally by the HPC communication library, and are seldom reported to the end-user. Partial disconnect of the network is a very difficult condition to

correct or even diagnose (for a particular PE, it may appear as if some other alive PE has been the victim of a crash failure). Possible resolutions involve routing around the problematic links, or promoting the link failure to a crash failure of the non-reachable processes. Even when the OpenSHMEM library resolves a link failure internally with rerouting, the potential for a severe reduction in performance motivates reporting an error to the application, to interrupt the normal execution flow and inform the user about the condition. Given supplementary introspection capabilities of the network topology, a maleable application may choose to adjust its communication pattern to match the new network capacity, or abort orderly if it is not maleable or when the performance loss is deemed too severe.

Corruption Failures: The last class of failures are referred to as byzantine failures. In this class of failures, affected processes may behave erratically, including malicious and intentionally disruptive behavior [18]. Although this class is generally intractable in asynchronous distributed systems, given reasonable assumptions about the type of erratic behavior, for example limited to dataset (not program) corruption [15], a variety of detection and mitigation strategies can be deployed. Beyond the protection provided by ECC memory, the detection of a silent error, and often the correction strategies are highly algorithm and/or dataset dependent [4,11] and cannot be detected or managed by the OpenSHMEM library. However, it may be desirable for an application detecting such an erroneous condition to receive support from the OpenSHMEM library in order to trigger a "recovery action" with other PEs.

3 Scope and Locality of Error Reporting

3.1 Local Versus Global Error Reporting

First, as we have discussed above, many of the failure classes that are the root cause for reporting errors are local to a PE, or are detected locally by some PE. Meanwhile other PEs have no chance to even observe the erroneous behavior. In a limited number of cases, *e.g.,* for some resource exhaustion errors, the PE triggering the error may be able to correct the error independently. There is therefore no strong case for alerting other PEs of the condition, as it will soon be corrected without their involvement, or knowledge. In many cases, however, some failures have to be reported at multiple, potentially all, PEs. In the case of a collective synchronization operation, for example, when a crash failure happens at a PE before it enters the operation, other PEs cannot possibly synchronize, and will have to report an error. The one-sided nature of many OpenSHMEM operations can also force reporting a failure at multiple PEs, without a direct mapping between the failed PE and which PEs have to report the error. Consider the case described in the left of Fig. 1, where P_1 issues a `shmem_wait` operation. This operation blocks until the remote updates performed from remote PEs toggle a conditional statement on the value. The origin PE (or PEs) that perform the remote updates are not specified by the operation. Consequently, if a process crash failure happens, the communication library cannot infer if one

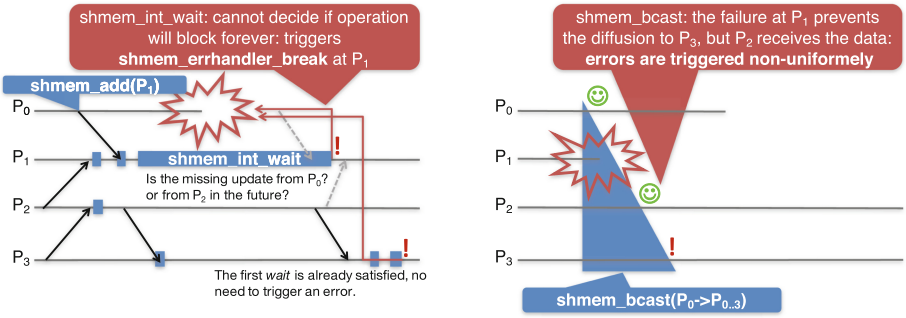


Fig. 1. Scope and uniformity semantics for error reporting. On the left, errors are reported locally, only for operations that are at risk of blocking indefinitely. On the right, a failure results in non-uniform errors: some PEs complete the broadcast, unaware that other PEs have triggered an error during the same collective communication.

of the failed PEs (here P_0) was supposed to perform the needed update, or if another PE (for example P_2) is soon going to post the update. In order to avoid leaving the target PE blocking in the posted `shmem_wait` operation indefinitely, the OpenSHMEM library has to report an error, ending that operation. However, other PEs may be able to satisfy all their blocking operations independently, and an error may be delayed until an operation would block. An advantage of this approach is that PEs that do not need to block (the second `shmem_wait` at P_3, for which the update has already happened) can spare the cost of checking for errors in the performance critical, non-erroneous execution path, unless an operation effectively blocks.

The general semantic is that error reporting is local, mandated to happen only at PEs whose completion of a blocking operation is rendered impossible by a failure (possibly multiple PEs, if they had issued a collective operations or `shmem_wait` operations), and is by default not propagated. However, we observe a dichotomy in use-cases. Some errors, for example resource exhaustion and some *soft* failures, can be easily corrected locally, and the local reporting permits maximal performance in that case. Some errors demand a collective correction action, and the proposed OpenSHMEM interface needs the capability to report errors both locally or globally. We will further discuss how global reporting can be triggered in Sect. 5.

3.2 Non-uniform Error Reporting

Conserving a strongly consistent global state, even after an error has been reported, is a very natural desire for application programmers. In a distributed system, providing such a strong semantic is unfortunately rife with multiple caveats. Even considering that some errors may trigger on a global scope, at all PEs, performance considerations still discourage providing uniform error reporting.

First, let's further define what it means for an error to be uniformly reported. An error is uniformly reported when all PEs get the notification of the error *at the same time*, that is, if a PE observes an error at a particular point in the program life, it can infer when a similar error has been triggered at target or origin PEs according to the Lamport causal ordering of communication operations in the program [17]. In collective and two-sided operations, there is a clear semantic linkage between the matching operations that form a line where one can easily define what uniform reporting means. In one-sided communication, such a clear operation based causality line is absent from the source code at the target, but one can still define a semantic line between the operation that failed at the origin, and the failure for the specified behavior to manifest at the target (*i.e.,* an origin performs an `shmem_add`, but the value is not updated at the target due to a failure of some sort).

Second, let's observe the performance implications of uniform reporting on OpenSHMEM operations. Consider, for example, the case of reporting errors during a `shmem_bcast`, as illustrated on the right of Fig. 1. When the operation completes at a non-root PE, the `shmem_bcast` specification states that the destination array contains the broadcast values. However, it does not give any information about the state of the completion of the broadcast at other PEs (it actually explicitly forewarns that reusing the `Psync` argument in another call may require a separate, explicit synchronization). In essence, the cost of the broadcast operation does not include the cost of synchronizing. In many implementations, a broadcast will leverage the relaxed semantic to optimize the operation with a tree topology. In such an implementation, the broadcast is complete at PEs high in the tree (that is, closest to the root) long before the broadcast completes at leaf PEs. Without further modification, this can result in potentially non-uniform triggering of errors, with some PEs reporting that the operation succeeded while other PEs report that it failed. With the added requirement that any error reported at a leaf PE must be consistently observed as an error reported at all other PEs, the overall cost of the broadcast then increases. The operation becomes semantically equivalent to an all-to-all operation (where each process contributes with the error code value), whose minimal cost is that of an AllReduce. That cost is present even when there is no error to report. Furthermore, if a PE fails *during* a synchronizing operation (that is, after it started contributing to the collective call), the failed PE could have passed its contribution to only a subset of its neighbors (in the topology used internally by the library). If the remaining PEs have to report uniformly that the operation has failed, the synchronization has to operate between non-failed PEs to agree, in a fault tolerant fashion, what the operation should report at all PEs. In practice, a fault tolerant synchronization (an agreement on a single value) can be twice as expensive as an AllReduce [16].

Similar to the case of collective operations, a strong mandate for reporting errors at the origin for any violation of the semantic at the target requires synchronizing all one-sided operations. The difference between the `shmem_fadd` and `shmem_add` operations is a prime exhibit of the cost of this implicit

synchronization with the target. The former returns the result of the operation at the origin, while the later does not, henceforth sparing the semantic synchronization with the target. These two operations have been separated, because the addition of this synchronization semantic has a salient impact on injection rate and latency performance of one-sided operations.

For these reasons, uniform error reporting is not required from OpenSHMEM operations. Instead, users are provided with additional interface to resynchronize PEs after an error has been reported. We will see in Sect. 6 how additional OpenSHMEM interfaces can help users in creating error handling epochs that ensure a clear discrimination between errors arising before and after the epoch starts.

4 Error Reporting Interface

In this section, we present the interface that embraces the principles exposed above, with some discussion about alternative software engineering designs that have been considered but rejected.

Error Handlers. Most OpenSHMEM operations may report errors. Errors can originate from invalid arguments being passed to OpenSHMEM operations, or from unexpected runtime conditions such as a processor or a network link failure, resource exhaustion, etc. Errors are reported by the invocation of the *error handler* associated with the error code (Fig. 2 presents a list of error handler management functions). The default error handler is set to `shmem_errhandler_gexit`, a predefined error handler that calls *shmem_global_exit*, thereby ending the entire application. This behavior is consistent with expectations of non error-managing OpenSHMEM applications. A program that manages errors should set an appropriate error handler, using the `shmem_errhandler_set` function, for each error code it can handle (or for all errors when using the special error code `SHMEM_ERR_ALL`). The error handler can be set with a predefined error handler (see Table 1 for the full list), or with an user provided function that receives the error code as input. Setting an error handler is a local operation, and each PE may set a different error handler for the same error code.

```
1  typedef void (*shmem_errhandler_cb_fn)(int errcode, void* user_params);
2
3  void shmem_errhandler_set(
4    int errcode,                  /*  IN: the managed error type */
5    shmem_errhandler_cb_fn errh, /*  IN: the error handling function */
6    void* user_params);           /*  IN: an user parameter to the callback */
7
8  void shmem_errhandler_get(
9    int errcode,                  /*  IN: the managed error type */
10   shmem_errhandler_cb_fn errh, /* OUT: the currently set error handler */
11   void* user_params);           /* OUT: the currently set user parameter */
```

Fig. 2. C Interfaces to manage error handlers in OpenSHMEM.

Table 1. List of predefined error handlers in OpenSHMEM.

`shmem_errhandler_gexit`	The error handler calls *shmem_global_exit* with the error code as parameter, which effectively terminates the application. This is the default error handler
`shmem_errhandler_break`	The error handler breaks from blocking OpenSHMEM operations at the PE. It has no effect at other PEs
`shmem_errhandler_gbreak`	The error handler breaks from blocking OpenSHMEM operations at all PEs

Rationale: During the design phase of the interface, alternative approaches where considered. Using return codes from OpenSHMEM functions would require to add a non-void return from most of the API functions. However, some operations, like `shmem_fadd`, already return a semantically important value from the function (the value of the target variable at the remote PE), which would have rendered that API change non-backward compatible. Another alternative, the use of a global `shmem_errno` value, was also considered. But this approach would entail difficulties for thread-safe operations in multithreaded programs. In addition, a programming style where the user has to check errors after all OpenSHMEM library calls was deemed to impose a high productivity tax on users, and for all these reasons, a reactive approach based on error handling callbacks has been preferred.

When an Error Handler Triggers. Implementations are encouraged to report the occurence of failures by triggering the local error handler function, with an appropriate error code, and strive not to leave any PE blocking in an operation disrupted by a failure. However, depending on the severity of the failure, it may not always be possible to do so (for example, in the case of a byzantine failure). Passing invalid arguments to OpenSHMEM operations generally results in undefined behavior; however, a debugging version of an OpenSHMEM implementation may check for invalid arguments and report errors.

When a user-provided error handler function returns, it has the same effect as if it had called `shmem_errhandler_break` as its last statement, that is, it interrupts ongoing OpenSHMEM communication blocking calls at the local PE. After an error handler has been triggered, OpenSHMEM communication operations do not block, and possibly do not respect their specification. That is, a synchronizing operation may return before synchronizing, or the data objects could be partially or incorrectly updated. Implicit non-blocking operations originating at the PE are also interrupted. It should also be noted that, due to the one-sided nature of OpenSHMEM operations, when an error is reported at an origin PE, incorrect behavior may also be observed at the target PEs without that PE reporting an error. It is possible to force an error to be reported at all PEs by calling the predefined error handler `shmem_errhandler_gbreak`, described in Sect. 5.

After an error has been reported, communicating with the OpenSHMEM library may not be possible. However, the memory allocated for symmetric data objects remains available at the local PE, giving the application a chance to verify the correctness of the data, take checkpoints before exiting, or continue using a resilient communication library. Operations that restore the communication capability of the OpenSHMEM library are described in Sect. 6.

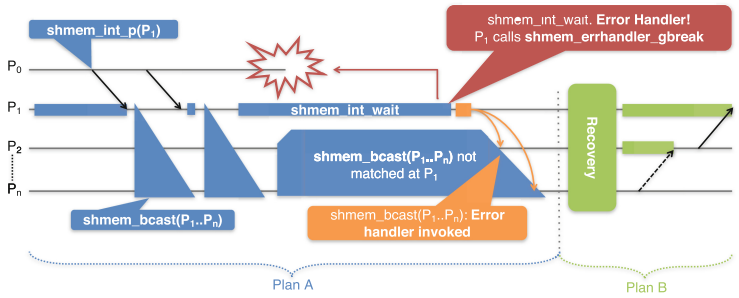
Stacking of Error Handlers. The user may call an error handler at any time, as a normal C function (including the predefined error handlers). In particular, a user defined error handler function can call another error handler function. In order to call the currently set error handler, a user can obtain the error handler and its parameter with `shmem_errhandler_get`, and can then call that error handler directly, or set the error handler with its own, and chain the call from within the replacement error handler. Similar interfaces are provided for Fortran, with the addition of an interface to call an error handler function.

Thread Safety. Although OpenSHMEM does not have complete definitions regarding thread safe operations at this point, we envision the following behavior with regard to error handler invocation in multithreaded programs. The error handler would be invoked once per PE. After the error handler would have been invoked, operation blocking at any thread of the PE would break. The apparent ordering of concurrent operations and error handler invocation would be implementation dependent.

5 Error Propagation

After an error has been reported to a particular PE, that PE may choose, or be constrained to stop performing operations and updates from the error free execution path. If the communication pattern is complex, the occurrence of failures can deeply disturb the application and, with only local error reporting, could prevent an effective recovery from being implemented. Consider the example in Fig. 3: as long as no failure occurs, the processes are following a communication pattern called *plan A*. PE P_0 does a `shmem_put` on a value at P_1. P_1 is blocking in a `shmem_wait` until that update from P_0 is made, then combines the result of the updated value with a local state, and broadcast that value to all other PEs, except for P_0.

Let's observe the effect of introducing a crash failure in *plan A*, and consider that P_0 has failed. As only P_1 blocks in an operation that could originate at P_0, other processes do not have to detect this condition, and only P_1 is guaranteed to have the failure of P_0 reported, as it issued a `shmem_wait` operation. The situation at P_1 now raises a dilemma: $P_{1..N}$ wait on the contribution of P_1 to the `shmem_bcast`. As all processes participating in the broadcast are alive (P_1 being a non-failed process), the operation may block until the matching `shmem_bcast` is posted at P_1. However, P_1 knows that P_0 has failed, and that the application




```
1  if(0==rank) {
2    shmem_int_p(&cond, 1, 1);
3    cond++;
4  } else {
5    if(1==rank) shmem_int_wait_until(&cond, comp++);
6    shmem_broadcast32(&comp, &comp, 1, 1, 1, 0, npes-1, psync);
7    /* (dest, src, count, root, PEstart, PEstride, PEsize, psync) */
8  }
```

Fig. 3. The transitive communication pattern *plan A*, from the source code, must be interrupted before the PEs can switch to the recovery communication pattern *plan B*. By calling the `shmem_errhandler_gbreak` error handler, P_1 ensures that all possibly unmatched operations in *plan A*, which could provoke deadlocks, are interrupted.

should branch into its recovery procedure *plan B*; if P_0 were to switch abruptly to *plan B*, it would cease matching the broadcast $P_{1..N}$ posted, following *plan A*. At this point, P_1 needs an effective way of interrupting operations that it does not intend to match anymore, otherwise, the application would reach a deadlock.

The proposed solution to resolve this scenario is that, before switching to *plan B*, the user code in P_1 sets the error handler to `shmem_errhandler_gbreak`, or explicitly calls `shmem_errhandler_gbreak` from within the user supplied error handler. The invocation of the predefined `shmem_errhandler_gbreak` error handler at any PE forces the invocation of the locally set error handler, with the same error code, at all PEs. As a consequence, communication operations do not block anymore and the OpenSHMEM library returns control to the user at all PEs, thereby solving potential transitive dependence deadlocks.

Implementation Challenges: An implementation has to be able to process the reception of a `shmem_errhandler_gbreak` notification. Some implementations use an asynchronous state machine to manage communication calls, and in these implementations, receiving the notification and interrupting ongoing operations is relatively simple. For implementation that employ blocking transport calls, different options are available. The implementation may employ a service thread to poll for `shmem_errhandler_gbreak` notifications and externally cancel blocking transport calls, or it may employ timeouts to interrupt blocking transport calls when their duration is excessive, and poll for notification only in this case. Ideally, polling for notification should be a low priority task, and the

specification permits delaying error notification after any latency and injection rate critical operations have completed.

As this operation aims at managing error cases, it has to itself tolerate the failures it reports. As such, this operation is not interrupted by normal errors. In particular, in an OpenSHMEM implementation that can tolerate crash failures, it has to perform a reliable broadcast to all surviving PEs. Fortunately, efficient implementations of a similar operation in fault-tolerant MPI exist (`MPI_Comm_revoke`), and have been demonstrated to be scalable [8].

6 Post-error Stabilization

At this point, the proposed interfaces have permitted reporting errors for locally observed failures, propagating these errors to all PEs in order to interrupt the code flow and regroup in a recovery procedure, but these interfaces have not permitted resuming OpenSHMEM communication after an error handler has been invoked.

One of the difficult points in resuming communication is determining that all PEs are aware of the same set of erroneous conditions. As described in Sect. 3.2, some errors may have been reported only at some PEs. Even when these PEs have triggered a global propagation with `shmem_errhandler_gbreak`, the notification of these propagated errors have communication delays, and may be observed at different causal times at different PEs. In order to stabilize the state of the application, the user needs to have an operation that (1) drains pending error notifications and ensures that the propagation of `shmem_errhandler_gbreak` notifications have completed, and (2) restores the communication capabilities between a globally agreed upon set of PEs that report a good health state.

The `shmem_error_barrier_all` provides these two capabilities in OpenSHMEM. It is a collective operation that provides a fault tolerant barrier between all non-failed PEs, which quiets all communications, and enforces that `shmem_errhandler_gbreak` propagation have completed. If, at a PE, the invocation of the `shmem_errhandler_gbreak` error handler precedes the call to the `shmem_error_barrier_all`, then, the local error handler is invoked at all PEs before the call completes. The error handler may be invoked from within the `shmem_error_barrier_all` without interrupting the operation, and it is the users' responsibility to ensure that the error handler does not call recursively `shmem_error_barrier_all`. The `shmem_error_barrier_all` operation completes in the presence of the failure types the OpenSHMEM implementation can tolerate, that is, the operation will block until an agreement is made that all the necessary error handlers have been invoked, and that the status of failed PEs has been agreed upon.

When the `shmem_error_barrier_all` operation completes, the status of PEs can be queried with the new local operation `shmem_error_query`, which, for the same PE argument, returns the same error status at all querying PEs. If a PE continues to be in a failed state, a query of its status returns the error code

representing the type of failure preventing the PE from continued participation in OpenSHMEM, or the special status 0 when the PE is capable of resuming communication with OpenSHMEM. Note that the status of a process changes only when `shmem_error_barrier_all` is called. A PE may query its own status, which may report that it cannot use OpenSHMEM anymore. In this case, the PE may initiate an orderly termination for itself, take checkpoints, or resort to an alternate communication library (such as MPI) to continue the parallel application.

Communications targeting a PE in error status trigger an error at the origin. Collective operations are collective over the subset of PEs that do not have an error status. PE ranks, the size of the pSync array, and offsets in data buffers remain unchanged. The content of the source and destination buffers that would have been sent or received from a PE in error status is unused.

7 Related Work

Fault tolerance and error reporting in communication middleware has a long history. The UNIX Socket interface is notably resilient to many failure types, and has the ability to report errors to endpoints on a socket. One of the main differences, which simplifies greatly the problem, is that sockets are bidirectional connected streams between two participants. In HPC communication libraries, managing an error not only means that the two endpoints of a failed stream are informed, but that mechanisms are in place to unblock all processes of the application that may risk blocking in multipartite communication operations, and globally establish a recoverable application state. Also, performance consideration are more stringent, as zero-copy and one-sided operations leave little opportunity to hide the cost of failure detection activities.

MPI faces many of the same distributed system challenges as OpenSHMEM, and has long provided the capacity of reporting errors. Efforts to define in the standard a recoverable state after MPI errors is however fairly recent, considering mostly crash-failures [5]. In two-sided MPI operations the participants to the operation are usually well specified (receives from named sources, etc.), which has permitted the fault tolerance specification to strictly scope which communication operations are interrupted when an error is reported. As a consequence, resilience extension in MPI are very operation centric and provide only explicit error reporting propagation. In contrast, the OpenSHMEM interface observes that many 1-sided operations do not specify clearly the origin, henceforth OpenSHMEM provides both explicit and implicit error propagation.

GASPI [23] is another PGAS communication library which features error management capabilities. Unlike in OpenSHMEM, all operations in GASPI have a timeout, after which they stop blocking (even when the operation has not completed). GASPI then provides explicit failure detection and observation routines to detect crash failures. In contrast, this fine grain handling is internal to the OpenSHMEM library, which returns from blocking operations only when the

implementation has observed that a failure (not necessarily limited to a crash-failure) may result in the operation blocking indefinitely, therefore simplifying the error management code.

Global View Resilience (GVR) [24] is a PGAS programming model that provides resilience to failures (bit flips, crash, etc.) with a resilient storage of multiple versions of the dataset. Distributed array can be streamed concurrently, and independently to the resilient storage, which keeps an history of multiple versions. Callbacks permit reconstructing damaged dataset when applicable. The extensions proposed in OpenSHMEM are orthogonal to the advanced abstraction of checkpointing proposed in GVR, which may benefit from resilience capabilities in OpenSHMEM to accelerate its own communications.

8 Conclusions and Future Work

In this work, we explore the addition of error semantics to the OpenSHMEM specification, and how one can leverage these constructs to recover from unexpected runtime errors and resource failures. The proposed interface is carefully crafted to preserve performance, avoiding the pitfalls of uniform or global error reporting. Instead, end-users are provided with the means to express their preference regarding the scope of reporting (global or local), and can restore the consistency of the application's global state after an error has been reported, by employing an easy to understand error barrier construct.

Overall, the designs makes OpenSHMEM capable of managing many failure vectors and resource exhaustion conditions by deferring the ultimate recovery action to the end-user, which can then try to stabilize the application and resume OpenSHMEM operations, or may fallback to an alternative interoperable communication interface to complete the application in a degraded mode.

At this point, the interface does not support spawning replacement PEs in stead of PEs in an unrecoverable state (wether they have encountered a hardware or crash failure, or a non-crash failure has rendered the state of the software stack unsafe to recover from). Many applications are not malleable, and require a fixed number of PEs. Thus, future works should explore extensions to this interface that permit replacing the failed processes, or, as an alternative, cooperate with an external mechanism (such as PMIx [2], or a fault tolerant MPI [5], etc.) to spawn the needed replacement PEs.

References

1. Amarasinghe, S., et al.: Exascale programming challenges. In: Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA. U.S Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), July 2011. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/ProgrammingChallengesWorkshopReport.pdf

2. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Krishna, J., Lusk, E., Thakur, R.: PMI: a scalable parallel process-management interface for extreme-scale systems. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 31–41. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15646-5_4. http://dl.acm.org/citation.cfm?id=1894122.1894127
3. Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: FTI: high performance fault tolerance interface for hybrid systems. In: International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2011 (2011)
4. Benoit, A., Cavelan, A., Robert, Y., Sun, H.: Assessing general-purpose algorithms to cope with fail-stop and silent errors. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) PMBS 2014. LNCS, vol. 8966, pp. 215–236. Springer, Heidelberg (2015). doi:10.1007/978-3-319-17248-4_11
5. Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: Post-failure recovery of MPI communication capability: design and rationale. Int. J. High Perform. Comput. Appl. **27**(3), 244–254 (2013). http://hpc.sagepub.com/content/27/3/244.abstract
6. Bosilca, G., Bouteiller, A., Guermouche, A., Herault, T., Sens, P., Robert, Y., Dongarra, J.J.: Failure detection and propagation in HPC systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016. ACM, New York (2016, to appear)
7. Bosilca, G., Bouteiller, A., Brunet, E., Cappello, F., Dongarra, J., Guermouche, A., Herault, T., Robert, Y., Vivien, F., Zaidouni, D.: Unified model for assessing checkpointing protocols at extreme-scale. Concur. Comput. Pract. Exp. **26**(17), 2772–2791 (2014). doi:10.1002/cpe.3173
8. Bouteiller, A., Bosilca, G., Dongarra, J.J.: Plan B: Interruption of ongoing MPI operations to support failure recovery. In: Proceedings of the 22nd European MPI Users' Group Meeting, EuroMPI 2015, pp. 11:1–11:9 (2015). http://doi.acm.org/10.1145/2802658.2802668
9. Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.J.: Correlated set coordination in fault tolerant message logging protocols for many-core clusters. Concur. Comput. Pract. Exp. **25**(4), 572–585 (2013). doi:10.1002/cpe.2859
10. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM (JACM) **43**(2), 225–267 (1996)
11. Chen, Z.: Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods. In: Proceedings of the PPoPP, pp. 167–176 (2013)
12. Davies, T., Karlsson, C., Liu, H., Ding, C., Chen, Z.: High performance linpack benchmark: a fault tolerant implementation without checkpointing. In: Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011). ACM (2011)
13. Dongarra, J., et al.: The international exascale software project roadmap. Int. J. High Perform. Comput. Appl. **25**(1), 3–60 (2011). doi:10.1177/1094342010391989
14. Ferreira, K., Stearley, J., Laros III, J.H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P.G., Arnold, D.: Evaluating the viability of process replication reliability for exascale systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 44:1–44:12. ACM, New York (2011). http://doi.acm.org/10.1145/2063384.2063443
15. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: Proceedings of the SC 2012, p. 78 (2012)

16. Herault, T., Bouteiller, A., Bosilca, G., Gamell, M., Teranishi, K., Parashar, M., Dongarra, J.: Practical scalable consensus for pseudo-synchronous distributed systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, pp. 31:1–31:12. ACM, New York (2015). http://doi.acm.org/10.1145/2807591.2807665
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
18. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982). doi:10.1145/357172.357176
19. Moody, A., Bronevetsky, G., Mohror, K., de Supinski, B.R.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2010). http://dx.doi.org/10.1109/SC.2010.18
20. Petrini, F., Frachtenberg, E., Hoisie, A., Coll, S.: Performance evaluation of the quadrics interconnection network. Cluster Comput. **6**(2), 125–142 (2003). doi:10.1023/A:1022852505633
21. Poole, S.W., Hernandez, O.R., Kuehn, J.A., Shipman, G.M., Curtis, A., Feind, K.: OpenSHMEM - toward a unified RMA model. In: Padua, D.A. (ed.) Encyclopedia of Parallel Computing, pp. 1379–1391. Springer, Heidelberg (2011)
22. Schroeder, B., Gibson, G.: Understanding failures in petascale computers. J. Phys.: Conf. Ser. **78**, 12–22 (2007). IOP Publishing
23. Shahzad, F., Kreutzer, M., Zeiser, T., Machado, R., Pieper, A., Hager, G., Wellein, G.: Building a fault tolerant application using the GASPI communication layer. In: Proceedings of the 2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, pp. 580–587. IEEE Computer Society, Washington (2015). http://dx.doi.org/10.1109/CLUSTER.2015.106
24. Zheng, Z., Chien, A.A., Teranishi, K.: Fault tolerance in an inner-outer solver: a GVR-enabled case study. In: Daydé, M., Marques, O., Nakajima, K. (eds.) VECPAR 2014. LNCS, vol. 8969, pp. 124–132. Springer, Heidelberg (2015). doi:10.1007/978-3-319-17353-5_11