GVR Documentation

Release 1.0.0

GVR Group <gvr@cs.uchicago.edu>

October 05, 2014

CONTENTS

CHAPTER

ONE

RELEASE NOTES

1.1 What is GVR?

GVR (Global View Resilience) is a user-level library that enables portable, efficient, application-controlled resilience. The primary target of GVR is HPC applications that require both extreme scalability and performance as well as resilience. GVR's key approaches include independent versioning of application arrays, efficient partial or whole restoration, open resilience to maximize the number of errors that can be handled (minimize fail-stop occurrences). Application knowledge can be exploited to control overhead, maximize error coverage, and maximize recoverable errors.

More information of GVR project is available at http://gvr.cs.uchicago.edu/

GVR has been developed by University of Chicago and Argonne National Laboratory, under the lead of Prof. Andrew A. Chien and Dr. Pavan Balaji. It has been supported by the U.S. Department of Energy, Office of Science / ASCR under awards DE-SC0008603/57K68-00-145.

1.1.1 Features

- Portable application-controlled resilience and recovery with incremental code change
- Versioned distributed arrays with global naming (a portable abstraction)
- Reliable storage of the versioned arrays in memory, local disk/SSD, or global file system
- Whole version navigation and efficient restoration
- Partial version efficient restoration (incremental "materialization")
- Independent array versioning (each at its own pace)
- Open Resilience framework to maximize cross-layer error handling
  - application-defined error handling
  - unified application and system error descriptors
  - attribute based composition for easy extensibility at application, operating system, and hardware levels
- C native APIs and Fortran bindings

1.1.2 Software and Platform Requirements

- Requires only an MPI library which is compatible with MPI-3 standard.
- Standard "autotools" preparation

- Requires no root privilege
- Runs on several platforms including x86-64 Linux cluster, Cray XC30 and IBM Blue Gene/Q

1.2 What's new in this version

1.2.1 1.0.0

- Introduced new Open Resilience error signaling & handling framework
- Implemented persistent data support, both in memory and on persistent storages (via the Scalable Checkpoint/Restart library)
- In addition to Linux/x86, also supported Cray XC30 and Blue Gene/Q platforms.
- Implemented several APIs
  - Implemented *GDS_create* function.
  - Implemented *GDS_get_acc* function.
  - Implemented *GDS_compare_and_swap* function.
  - Implemented *GDS_access*, *GDS_get_access_buffer_type*, *GDS_release* functions.
  - Supported *label* and *label_size* arguments in *GDS_version_inc* function.
- Updated documentation and added program examples.

1.2.2 0.9.1

- Sub-communicators (communicators which are subset of *MPI_COMM_WORLD*) are now supported in *GDS_alloc*.

1.2.3 0.9.0

- Removed dependency to CUnit from test suite.
- Size type of a communicator became *int* (originally *size_t*).
- Implemented *GDS_get_comm* function.
- Implemented *GDS_compare_and_swap* function.
- Implemented parameter sanity checking for basic functions. Set an environment variable *GDS_SANITY_CHECK* to 1 to activate.

1.2.4 0.8.2-rc0

- Introduced sleep scheme switching infrastructure (*GDS_WAIT_SCHEME*)
- Fixed unnecessary memory consumption around dynamic tag system
- Target buffer is now zero-cleared on allocation
- Several improvements to log-structured array
  - *GDS_acc* support

1.2.5 0.8.1-rc0

- Added log-structured array implementation
- Several performance improvements in the UChicago RCC Midway cluster (MVAPICH2-2.0b + Infiniband).

1.2.6 0.8.0

- Added Fortran APIs, and related documentation.
- Add *GDS_register_global_error_check* and *GDS_register_local_error_check*, and the test case.
- Separate the error descriptor enqueue function from raise error function.
- Supported *min_chunks* argument in *GDS_alloc*.
- Supported self version description. To show the version of the GVR library, set the environment variable *GDS_SHOW_INFO* to 1.

1.2.7 0.7.1

- Fixed a bug that *tests/dprint.h* was not included in the release tarball.

1.2.8 0.7.0

- Add local error polling and handling for the APIs including *GDS_get/put*, *GDS_acc*, *GDS_wait*, *GDS_fence*, *GDS_descriptor_clone*, and *GDS_version_inc*.
- Set a flag of recovery mode. If the flag is on, they will not reentrant the error handling function.
- For each GDS object, global error recovery functions are invoked one by one when *GDS_fence* is called with *NULL*.

1.2.9 0.6.0

- Implements API changes in GVR API 0.7.3 and 0.7.4
  - Supports both row-major ordering and column-major ordering for multi-dimensional arrays. Default ordering for C binding is now row-major (same as in GA and C language itself).
  - Supports functionality to create error descriptor and error category.
- Supports functionality for global error handling, including global error handler registration, raising global error, and invoking global error handler.
- Supports functionality for local error handling, including local error handler registration, raising local error, and invoking local error handler.
- Array accessing primitives (get/put/acc) now become non-blocking, as defined in the API document. This may break some existing applications which do synchronize properly.
- New documentation system

1.2.10 0.5.3

- Fixed a misconfiguration that internal header files were missing from the tarball

1.2.11 0.5.2

- Fixed a bug that led to data corruption when restoring data bigger than 4KB from an old version (Issue #18)

1.2.12 0.5.1

- Fixed a bug that *gds.h* included unnecessary file, which led to a build failure

1.2.13 0.5

- Initial internal milestone
- Basic Functionality including ability to create global data structures, put/get, create versions, signal and handle errors, subject to restrictions listed below
- Implements GVR API version 0.72

1.3 Prerequisites

To install and use the GVR library, you need to have the following things properly installed and configured.

- Standard software development tools such as *make* or *gcc*. In Ubuntu, you can install them by doing *sudo apt-get install build-essential*.
- MPI library which supports MPI 3.0 standard. The *mpicc* command must be found in a directory included in the *PATH* environment variable.

We have been testing GVR on the following platforms. * Ubuntu 12.04 AMD64, GCC 4.6.3, and MPICH 3.1. * Midway at University of Chicago RCC (Scientific Linux 6.5, GCC 4.8.2, MVAPICH2-2.1a) * Edison at NERSC (Cray XC30) * Vesta at Argonne National Lab. (IBM Blue Gene/Q)

1.4 How to build & install

GVR can utilize LRDS (Local Reliable Data Storage) as an external memory error event reporter. Use of LRDS is optional. In order to configure GVR to use LRDS, first build and install LRDS.

Then, the installation process of GVR is straightforward by,

```
$ ./configure [OPTIONS]
$ make && make install
```

Some important parameters for the `configure` script:

- `--prefix=` specifies the installation target directory under which several sub-directories such as `include/`, `bin/`, and `lib/` are created.

- `--with-lrds=` specifies the installation directory for LRDS. If you specified a directory to `--prefix` when configuring LRDS, specify the same directory here. If `--with-lrds` is not specified, GVR will be configured and built not to use LRDS.

1.4.1 Platform support: Blue Gene/Q

On BG/Q machine, to build GCC toolchain and latest MPICH, please follow the instructions at https://wiki.mpich.org/mpich/index.php/BGQ.

To build GVR, specify host as *powerpc64-bgq-linux* during the configuration.

```
$ ./configure --host=powerpc64-bgq-linux [OPTIONS]
```

1.4.2 Platform support: Cray XC30

On Cray machine, to build GVR, specify host as *x86-cray-linux* during the configuration.

```
$ ./configure --host=x86-cray-linux [OPTIONS]
```

1.4.3 Optional library: SCR

GVR supports persistent data with SCR (Scalable Checkpoint Restart) library (http://sourceforge.net/projects/scalablecr/) developed by Lawrence Livermore National Laboratory (https://computation-rnd.llnl.gov/scr/). GVR utilizes SCR to store data across storage hierarchy (local file system, parallel file system, etc.), therefore provides stronger resilience towards severe failure cases such as process failures.

SCR installation is optional. GVR works with latest SCR version 1.1-7. To install SCR, please follow the install instructions inside the package.

Install GVR with SCR:

```
$ ./configure [OPTIONS] --with-scr=$(SCR_PATH)
$ make && make install
```

Before running GVR programs with SCR, you need to configure SCR. There are several environmental parameters.

- *SCR_CACHE_BASE* specifies the location in local file system for storing data. Please make sure there are enough space for storage.
- *SCR_PREFIX* specifies the location in shared file system for storing data.
- *SCR_CONF_FILE* specifies the detail configuration for SCR utilization.
- *SCR_JOB_ID* specifies the unique job number. Each instance of program should be associated with a unique number.
- *FLUSH_to_SCR* specifies the frequency to flush data to SCR.

Following example shows a configuration file and environmental variables settings for SCR.

```
$ cat myscr.conf
SCR_FLUSH=3
CACHEDESC=0 BASE=/tmp SIZE=12
SCR_COPY_TYPE=LOCAL
SCR_PREFIX=/home/DIR
```

```
SCR_LOG_ENABLE=0

$ export SCR_CONF_FILE=myscr.conf
$ export SCR_CACHE_BASE=/tmp
$ export SCR_PREFIX=/home/DIR
$ export SCR_JOB_ID=1
$ export FLUSH_to_SCR=10
```

1.5 Known issues and restrictions

Currently we have the following restrictions.

- Currently *GDS_THREAD_SINGLE* is the only supported thread execution model.
- Due to a potential bug, GVR may not work correctly on MVAPICH2 + Infiniband environment.
- *GDS_get_attr* does not support *GDS_GLOBAL_LAYOUT* attribute yet.

1.6 Contributors

GVR has been developed by University of Chicago and Argonne National Laboratory, under the lead of Prof. Andrew A. Chien and Dr. Pavan Balaji. Andrew led the entire team and designed basic architecture of the GVR. Pavan led Argonne side of the team and designed LRDS as well as basic GVR API set.

Several researchers and students from both University of Chicago and Argonne National Laboratory have been contributed to the project as core contributors. Hajime Fujita led the implementation effort and designed detailed system architecture of GVR and implemented many part of the core GVR functionalities such as array allocation/deallocation, basic data access functions. He also designed and implemented the Open Resilience error handling framework. Zachary Rubenstein implemented the initial prototype of GVR and later some of the core functionalities. He was also engaged in application studies regarding linear solvers and libraries, such as Trilinos and PCG solver. Aiman Fang designed and implemented the persistent data support in GVR, including utilizing the Scalable Checkpoint/Restart library from GVR. She also worked in application study for molecular dynamics (miniMD and ddcMD) applications. Ziming Zheng designed and implemented the basic error signaling and handling framework in GVR. He also developed the GVR-augmented Trilinos and worked on fault-tolerant study on linear solvers. Nan Dun worked on OpenMC application study, and GVR implementation around core APIs and building/testing system. Kamil Iskra designed and implemented LRDS, and also contributed to the Open Resilience design as an expert of system software. Wesley Bland contributed to the team as an expert of fault-tolerance in MPI. Chaofan Chen implemented the non-blocking version of the core data access functions.

Following members also contributed to the project. Pete Beckman, James Dinan, Jeff Hammond, and Robert Schreiber participated in the initial architectural design of GVR, and also gave several technical advice on its design and implementation strategies. Guoming Lu built an analytical model to show an effectiveness of multi-version snapshot under latent errors. Andrew R. Siegel provided insights from the view point of OpenMC. Bruno Cabral implemented a prototype of redundant data store support. Henry Riordan worked on enhancing Trilinos support.

Also we received plenty of useful suggestions and comments from our collaborators, on application studies: Ignacio Laguna, and David Richards for ddcMD, Anshu Dubey, and Brian van Straalen for Chombo, Mark Hoemmen, Mike Heroux, and Keita Teranishi for Trilinos and linear solvers, and John R. Tramm for OpenMC.

CHAPTER

TWO

INTRODUCTION

2.1 Concepts

- **Cooperative Resilience** Programmer cooperates with the system to manage application reliability. This includes providing information to guide resilience implementation. The programmer provides checking and recovery functions.
- **Versioned Arrays** Key abstraction is versions of arrays which represent the history of state evolution of the arrays. Versions should correspond to a stable (consistent) application point.
- **Resilience Priorities** Labelling of arrays as high, medium, or low resilience priority, which indicates to the system where effort/space should be expended to maximize return on application resilience.
- **Application Error Checks** Programmer-supplied routines that analyze arrays based on application semantics and, as appropriate, raise errors.
- **Application Error Handling** Programmer-supplied routines that analyze the error and the available array versions, and repair application and resume execution.

2.2 Objectives

- Enable programmers to express application-controlled resilience.
- Exploiting application semantics, multi-version arrays, and tolerating errors from hardware, software, unknown sources to continue execution.
- Application programmers able to control overhead of resilience and direct runtime effort with resilience priorities, and expression of version boundaries.

2.2.1 Open Issues

- Error-handling
  - Fault recovery asynchronous (error handler invoked asynch) or synchronous (error handler involved only at poll for faults).
  - Do we need any additional capabilities for error handling routines?
- Task-oriented parallelism
  - How to support unstructured task parallelism?

2.3 Design Requirements

- It should be possible to derive GVR code from traditional global array programs using only incremental translation. "Incremental translation" should be understood to mean local rewrites, such as macros.
- GVR is only guaranteed to work as expected on well synchronized programs. The semantics of GVR do not support race conditions.
- There should be no additional overhead *required* to use versioned GVR structures. That is, the semantics of GVR should support a single-version implementation.
- The application is expected to access older versions in the context of an error-recovery. It is assumed that applications will rarely be in this context.

CHAPTER

THREE

GETTING STARTED

3.1 Initialization

The simplest GVR program looks like this:

```
#include <gds.h>

int main(int argc, char *argv[])
{
    GDS_thread_support_t provd_support;
    GDS_init(&argc, &argv, GDS_THREAD_SINGLE, &provd_support);
    ...
    GDS_finalize();

    return 0;
}
```

3.2 Initialization with MPI library

If your program already uses MPI, you should call *GDS_init* **after** *MPI_Init_thread* and *GDS_finalize* **before** *MPI_Finalize*. MPI library must be initialized with *MPI_Init_thread* and *MPI_THREAD_MULTIPLE* for the thread support level.

```
#include <mpi.h>
#include <gds.h>

int main(int argc, char *argv[])
{
    int mpi_prov;
    GDS_thread_support_t provd_support;
    MPI_Init_thread(MPI_THREAD_MULTIPLE, &mpi_prov);
    GDS_init(&argc, &argv, GDS_THREAD_SINGLE, &provd_support);
    ...
    GDS_finalize();
    MPI_Finalize();

    return 0;
}
```

3.3 How to build your program

You need to link your GVR-applied program with *libgds* by specifying proper linking options, such as *-L* and *-lgds* options.

```
$ mpicc ... -I$GVR_PATH/include -L$GVR_PATH/lib <gvr-program>.c -lgds
```

For Fortran programs, you need to link with both *libgdsf90* and *libgds*. Make sure to specify -L, -lgdsf90, and -lgds options.

```
$ mpif90 ... -I$GVR_PATH/include -L$GVR_PATH/lib <gvr-program>.f90 -lgdsf90 -lgds
```

3.3.1 Note for static linking

If you want to link *libgds.a* statically to your program, you also need to link *lrds_dummy.a*. Note that *lrds.a* requires a special kernel feature and is unlikely to work on most of the systems.

```
$ mpicc ... -I$GVR_PATH/include -L$GVR_PATH/lib -static <gvr-program>.c -lgds -llrds_dummy
```

3.4 How to run your program

Current GVR implementation is built on top of MPI library, so you can run your GVR-enabled program just like an MPI application. You may have to add the library path to *LD_LIBRARY_PATH* where *libgds.so* is located.

```
$ LD_LIBRARY_PATH=$GVR_LIB_PATH mpiexec -n 4 <gvr-program>
```

3.4.1 Runtime Options

GVR library provides following environment variables to control the runtime behavior:

For debugging/problem reporting

- *GDS_DEBUG_LEVEL*: An integer variable to set verbosity of the debug output from the GVR library. Default is zero (only critical errors are reported) and the current maximum is 2 (all the debug messages will be shown). Debug output is written to the standard error.
- *GDS_SHOW_INFO*: If set to non-zero value, the GVR library will dump a banner at the beginning of the execution, including version information and all the compile time/runtime configurations. This will be useful for error reporting. Default is zero (disabled).
- *GDS_SANITY_CHECK*: If set to non-zero value, the GVR library will perform more careful check on input parameters. Default is zero (disabled).

For performance tuning

- *GDS_WAIT_SCHEME*: The GVR library uses a server thread to handle some of the internal service requests. This option controls how the server thread waits for incoming requests.
  - *0*: Use *MPI_Waitany*. This option is known to have a performance issue in many platforms.
  - *1*: Use a loop of *MPI_Testany* + short *usleep*.
  - *2*: Use a loop of *MPI_Testwany* + "CPU spinning" (empty loop).

- *GDS_WAIT_SLEEP_DURATION*: Set the sleep duration in microseconds if *GDS_WAIT_SCHEME* is set to 1.
- *GDS_WAIT_SPIN_COUNT*: Set the number of spin loop count if *GDS_WAIT_SCHEME* is set to 2.
- *GDS_OUTSTANDING_RMA_OPS_MAX*: In some platforms it is known that the large number of outstanding RMA operation is harmful for performance. This variable sets the maximum number of outstanding RMA operation per RMA window (=GVR array).

3.5 Code example

The *examples* directory contains two examples (in C).

- *dgemm.c*: A simple DGEMM program with GVR array creation and data access.
- *resilient_dgemm.c*: In addition to the above program, this also covers version creation, version navigation, error signaling and handling.

Writing Fortran programs requires users having basic knowledge of interoperability with C, especially dealing with C pointer and function pointer. For more details, please refer to GCC Interoperability with C. Note that the internal memory ordering of array is set to *column-major* by default for GVR Fortran library, which is different from *row-major* in GVR C library.

Following Fortran90 code shows an example of using GVR Fortran APIs. For more examples, please refer to *tests/tests_f90.f90* in GVR source directory.

```
PROGRAM FORTRAN_EXAMPLE()
  USE, INTRINSIC :: ISO_C_BINDING
  USE GDS
  IMPLICIT NONE

  TYPE(C_PTR) :: g ! GDS handler with C_PTR type
  INTEGER(8) :: ndim, cts(1), min_chunk(1), lo_idx(1), hi_idx(1), ld(1)
  INTEGER, TARGET :: put_buf, acc_buf, get_buf ! TARGET attribute is required for C_LOC
  INTEGER :: stat

  g = C_NULL_PTR
  ndim = 1
  cts(1) = 1
  min_chunk(1) = 0
  lo_idx(1) = 0
  hi_idx(1) = 0
  ld(1) = 0
  put_buf = 1
  acc_buf = 1
  get_buf = 0

  call GDS_ALLOC(ndim, cts, min_chunk, GDS_DATA_INT, GDS_PRIORITY_HIGH, &
        GDS_COMM_WORLD, MPI_INFO_NULL, g, stat)

  ! Use C_LOC to get buffer address
  call GDS_PUT(C_LOC(put_buf), ld, lo_idx, hi_idx, g, stat)
  call GDS_FENCE(g, stat)

  call GDS_ACC(C_LOC(acc_buf), ld, lo_idx, hi_idx, MPI_SUM, g, stat)
  call GDS_FENCE(g, stat)

  call GDS_GET(C_LOC(get_buf), ld, lo_idx, hi_idx, g, stat)
```

```
  call GDS_FENCE(g, stat)

  acc_buf = 2
  call GDS_ACC(C_LOC(acc_buf), ld, lo_idx, hi_idx, MPI_PROD, g, stat)
  call GDS_FENCE(g, stat)

  call GDS_GET(C_LOC(get_buf), ld, lo_idx, hi_idx, g, stat)
  call GDS_FENCE(g, stat)

  call GDS_FREE(g, stat)
END PROGRAM
```

CHAPTER

FOUR

OPEN RESILIENCE

4.1 Introduction

A computer system may encounter various kinds of errors during execution. Process or node crash is a typical example of such errors but also there can be a lots of other errors like data corruption in memory, disk, or network transaction. In addition to the large variety of the kinds of errors, errors come from various different layers. Errors could happen in and be detected by hardware, operating system, middleware for I/O or communication, numerical libraries, or applications. So in order to effectively deal with the real system, we need a framework to treat various kinds of errors from various layers.

One requirement for such a comprehensive error handling is a **cross-layer error handling**, which means multiple components in different layers coordinate together to handle errors. This is important because generally lower layer (e.g., hardware or operating system) is agnostic to higher-level (e.g., application) knowledge about data structures or properties, whereas these higher-level knowledge is a key for efficient error checking and recovery.

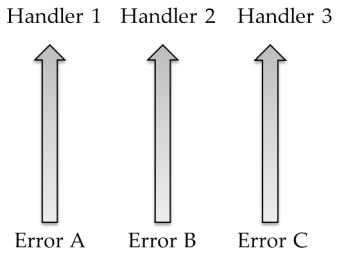


Figure 4.1: A traditional cross-layer error handling approach.

Traditional way of dealing with these many different errors in the cross-layer manner is the "siloed" approach, in which each error is tightly coupled with a specific error handler. Under an assumption that there are so many different types of errors, such approach is prohibitively expensive in terms of development effort. It also prohibits any reuse or generalization of existing error handlers. If an error handler is tightly coupled with a certain kind of error, it can not handle another kind of error which is quite similar to the originally expected one. In fact, some sort of errors can be treated in a similar way. For example, data corruption error can be recovered by a single handler regardless of the actual source of an error. This kind of **generalization** of an error handler can maximize the return of an investment to the development of error handlers. This will also encourage hardware/software vendors to invest more on rich error reporting. On the other hand, if a developer wants to have a new error handler very specific to a particular type of error events, one should be able to add the new one without removing or changing an existing, more generic handler. In other words, there's also a need for **specialization**.

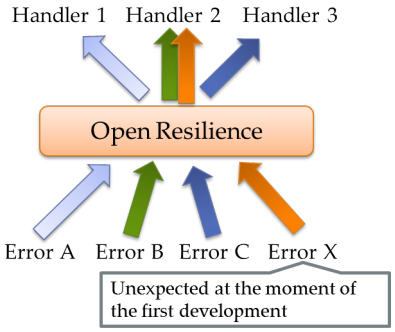


Figure 4.2: The Open Resilience approach for error signaling & handling.

GVR implements the **Open Resilience** framework to enable and encourage a synergy between error detecting side and error handling side, through the **unified error signaling & handling interface**. This section describes a general ideas and designs of the Open Resilience framework in GVR, followed by several motivating use case examples.

4.2 Design

Under Open Resilience, **error detectors**, which detect and report errors, and **error handler**, which receive error signals and handle errors, are first decoupled. Then GVR dynamically glues an error detector and an error handler through its unified interface, based on template matching of an error to handler. When an error detector detects an error, it signals an error by raising an error through the GVR interface. GVR looks up the most appropriate error handler among the registered error handlers, then invokes the chosen one. This approach improves composability and generality of error handlers, thus increasing leverage of error reporting and error handlers.

4.2.1 Error Attributes and Descriptor

An error needs several descriptive information, such as the size and location of a corrupted region or a list of failed processes. Errors in GVR are represented as a set of error attributes. Each error attribute is a pair of a key and a value. Some common error attributes are pre-defined by GVR but users can also define a new one.

A set of error attributes is called **error descriptor**. An error descriptor is a core object on error handling which is passed along different components to describe the details about an error. An error descriptor is created by *GDS_create_error_descriptor*, followed by *GDS_add_error_attr* to add error attributes.

4.2.2 Error Signaling and Handling

Local and Global Error Signaling & Handling

Some errors can be handled locally only by one process, while others may require coordinated error handling with multiple processes. In order to capture this difference, GVR introduces a notion of **local and global error signaling & handling**. Local error handling involves only one process, whereas global error handling involves all the processes in a group. The group usually is a set of users of one GVR array. If an error is not associated with any array, the group consists of all processes in the application.

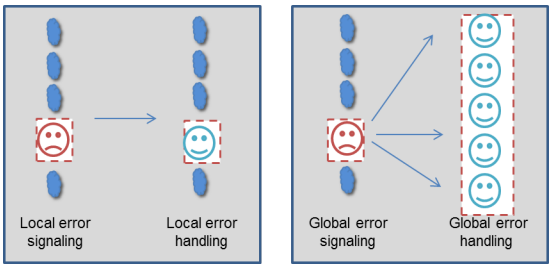


Figure 4.3: Global/local error signaling and handling.

Signaling Errors

When an error detector find an error, it first generates an error descriptor and store available information regarding the error as a set of error attributes. Then it raises an error using either of *GDS_raise_local_error* or *GDS_raise_global_error*, with passing the error descriptor.

Handling Errors

Errors are handled by error handlers. An error handler is a function which takes an error descriptor as an argument. An error handler is registered to a GVR array via either *GDS_register_local_error_handler* or *GDS_register_global_error_handler*. Each array can have multiple error handlers registered. When an error is raised by an error detector, GVR looks up the most appropriate error handler and invokes it.

Library-invoked Error Checking

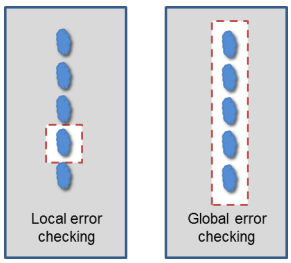


Figure 4.4: Global/local error checking.

A user can register his or her own error checking functions to GVR using *GDS_register_local_error_check* or *GDS_register_global_error_check*. GVR will invoke these functions based on at certain timing based on **error checking priority**. As in error signaling and handling, error checking also has a notion of local and global error checking. Local error checking is run by a single process whereas global error checking involves all processes which use a GVR array. Global error checking function is invoked in each process in a synchronized manner.

4.2.3 Matching between Errors and Error Handlers

Upon error event, GVR chooses the most appropriate error handler for a given error descriptor. This is because if multiple handlers were called, these handlers must have been designed carefully to make sure that they would not interfere each other, thus loosing generality and flexibility.

In order to choose the best handler automatically, each handler expresses its capability on error handling. More specifically, each handler is associated with an **error attribute predicate** to characterize what kind of error attributes it expects/handles. A predicate expresses a constraint on acceptable error descriptors.

A predicate is a set of terms (rules), where each term looks like one of the following.

1. **<KEY:*>**: An error attribute key *KEY* must appear in the descriptor, but its associated attribute value can be anything.
2. **<KEY:VAL>**: An error attribute key *KEY* must appear in the descriptor with a value *VAL*.
3. **<KEY:MIN-MAX>**: An error attribute key *KEY* must appear in the descriptor, and the value must be within the range between *MIN* and *MAX*.
4. **<KEY:!>**: An error attribute key *KEY* must *not* appear in the descriptor.

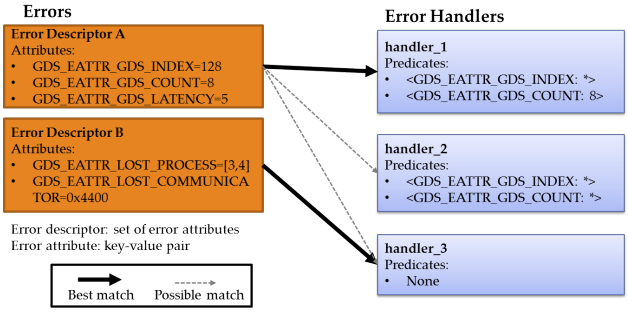
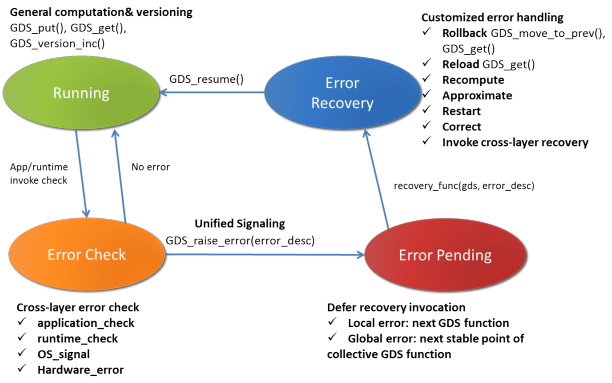


Figure 4.5: A matching example of errors and error handlers.

The above figure shows an example of matching between error descriptors and error handlers. When error descriptor A is raised, it can match with all the error handlers in the right-hand side, because all the rules in each handler are satisfied with the descriptor A. Then GVR has to choose the best one among these three. GVR regards the one with the most specific match as the best match. In this case *handler_1* has the most specific rules among three, so it will be invoked for the descriptor A.

To register an error handler, first a user has to prepare an error predicate. An error predicate object is created by *GDS_create_error_pred*. Then a user can call *GDS_create_error_pred_term* to create terms which correspond to matching rules. *GDS_add_error_pred_term* adds a term to an existing predicate object. Once a predicate object becomes ready, it will be passed to either of *GDS_register_local_error_handler* or *GDS_register_global_error_handler* with a function pointer to the handler.

4.2.4 Application Lifecycle



The above figure shows the typical application lifecycle regarding error handling. During the normal run (top left, "Running"), an application performs computation and communication. Also it stores critical data to a GVR array and creates versions occasionally.

During program execution, several error checks will be performed (bottom left, "Error Check"). Some checks are done by application itself, while others may be performed by other layers, such as hardware, operating system, communication middleware, numerical library, or GVR library. Once an error is found, an error is raised through GVR's unified error signaling interface. This error event (descriptor) is queued inside GVR and delivered to an appropriate handler at the next available timing (bottom right, "Error Pending"). For local errors, queued error will be delivered at the next GDS function, where as for global errors, queued error will be delivered at the next synchronization point (*i.e. GDS_fence* or *GDS_version_inc*).

Once an error descriptor is delivered to an error handler, it performs an error recovery based on its own strategy (top right, "Error Recovery"). During the recovery the handler can utilize data stored in the GVR array, not only in the latest version but also in an old version. Once a recovery is done, the handler calls *GDS_resume_local* or *GDS_resume_global* to tell GVR that error handling is done.

4.3 Example Scenarios

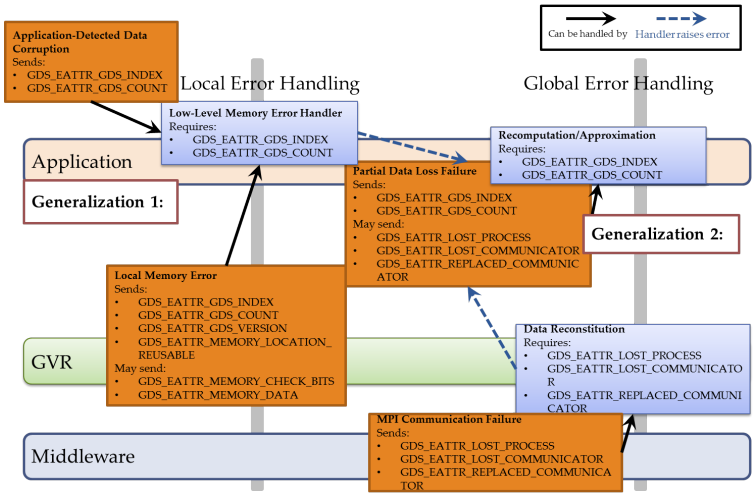


Figure 4.6: An error handling example scenario.

This figure shows some examples of generalization of error handlers under the Open Resilience error handling framework. The first example is the low-level memory error handler that handles a data corruption in a small region in a GVR array. This is a local error handler, meaning that it doesn't coordinate with other processes. It can handle two kinds of errors, one is a local memory error originally detected by a hardware and signaled through OS and GVR, and the other is a data corruption error detected by an application itself, using checksum for example. While these two errors come from different error detectors and possibly different causes, the handler can treat them in the same manner because both errors describes the corrupted location information with the same attributes, *GDS_EATTR_GDS_INDEX* (describes the location of the corruption) and *GDS_EATTR_GDS_COUNT* (describes the size of the corrupted region).

The second example shown here is the handler which performs recomputation or approximation to recover array data for data loss. The first use case of this handler is a case where the low-level memory error handler in the previous example figures out that it requires global coordination to complete the recovery. The second case is an MPI communication failure, typically caused by process or node crash. Both errors manifest as a data loss in a certain region, therefore this handler is able to handle both errors in the same fashion.

CHAPTER

FIVE

USE CASES

5.1 Introduction

GVR provides data-oriented resilience based on global view data and multi-version. It enables an application to incorporate resilience incrementally, expressing resilience proportionally to the application need. As a result, GVR fits for various types of parallel programs, including regular applications (SPMD) and irregular applications (task parallelism). Furthermore, GVR supports flexible usages, including direct application programming interface and co-existence with other runtime and programming model such as MPI.

In the use cases document, we first describe the typical parallel program structure from a data access point of view in order to build efficient global, reliable data structures. Then we present the use cases of general computation and multi-version checkpointing, followed by the examples of error check, signaling, and recovery.

5.2 Parallel program structure

Extensive research has studied design "patterns" for sequential computer programs and more recently for parallel programs. In general, study of design patterns has focused on patterns of interaction and composition from a program (generally function and control-centric) point of view. In sequential programs that presume a shared memory, these patterns may have little to do with program data access. In parallel programs, they are more closely related, but typically the design patterns are completely correlated (message passing interaction patterns and all other memory access local) or completely orthogonal.

We are interested in typical parallel program structure from a data access point of view in order to build efficient global, reliable data structures. In particular, we assume that the major application data structures are:

1. the primary basis of the parallelism
2. capture the critical sharing and coordination

We start from the perspective of models that have achieved high scalability and efficiency in large-scale parallel machines, working from simple cases to more complex ones. Global data structures which are read- only can also be important for large-scale parallelism, and their optimization can be performance critical. However, because they are simpler from a consistency and update point-of-view, we neglect them here.

5.2.1 A. Static, Regular Data Decomposition, Owner Computes

This model includes simple stencil, finite differences, SOR, etc. Computations with and without ghost regions. The regular data is decomposed for parallelism and remains in that configuration. The basic computational steps are:

1. Copy read set for computation to local
2. Compute (w/o communication)
3. Update write set locally (owner computes) 4. Synchronize globally (or with neighbors)

5.2.2 B. Static Irregular Data, Data Decomposition, Owner Computes

This model includes iterative methods on sparse structures. For example, irregular finite element graphs, sparse matrix iteration, irregular meshes, and includes specific examples such as NEK and OpenMC (codes important to CESAR). The irregular data is intelligently decomposed for parallelism, perhaps by a sophisticated graph-partitioning algorithm, and remains in that configuration. The basic computational steps are:

1. Copy read set for computation to local (irregular)
2. Compute (w/o communication)
3. Update write set locally (owner computes)
4. Synchronize globally (or with neighbors)

5.2.3 C. Irregular, Predictable Computation across Data (regular or irregular)

This model includes adaptive, iterative methods on sparse structures. For example, adaptive finite element graphs, adaptive irregular meshes, multigrid, and sophisticated n-body algorithms such as FMM and Barnes- Hut. The irregular data is intelligently decomposed for parallelism, perhaps by a sophisticated graph- partitioning algorithm. This decomposition may be a compromise across multiple phases (e.g. with multigrid or FMM), and after a number of iterations may be reorganized for load-balance/parallelism. The basic computational steps are: #. Copy read set for computation to local (irregular) #. Compute (w/o communication) #. Update write set locally (owner computes) #. Synchronize globally (or with neighbors) #. Check balance, and if necessary rebalance

5.2.4 D. Irregular, Unpredictable Computation across Data (regular or irregular)

This model includes unpredictable computational structures such as direct solvers. The challenge for these models is that the simpler data decompositions do not work well because either the relationship between data and computation are unpredictable (even if there is little), and the parallelism is irregular and dynamic, requiring ongoing rebalancing of computation across the machine. Typical structures used to manage are task parallelism, and a range of dynamic, adaptive load-balancing techniques. Here there are two classes of problems global data structures for managing the irregular parallelism (task queues, etc.) and to manage the shared data. Examples that fit here are NWChem Tensor Contraction Engine (TCE) that lies behind dozens of single- and multi-reference coupled-cluster methods such as CCSD. The elements of managing parallelism include:

1. Creation of tasks (enqueue)
2. Creation of workers
3. Claiming of tasks for execution

4. Completion of tasks and registration
5. Load-balancing (work-stealing, etc.)
6. Detecting termination of the phase (if well-defined)

Within each task, each worker typically undertakes the following steps with respect to the parallel work and shared data:

1. Copy read set for computation from global to local (regular or irregular)
2. Compute
3. Add tasks to the work pool(s)
4. Register completion and write results to global structures (regular or irregular)

What are important patterns of data sharing on the read/write sets?

5.2.5 E. Irregular, Unpredictable Computation across rapidly changing data (global shared memory)

The majority of computations at ALCF (and presumably other high-end computing centers)are dominated by classes A, B, and C. Computational chemistry methods are also an important class, so if they fall into category D, then all four of A-D are required for good coverage.

Lots of the new computing models for irregular problems are attacking things that are in category E. What fraction of the interesting scalable computations do these uncovered correspond to?

5.3 Case Studies: general computation and multi-version

Note: All algorithms are written from the perspective of a single process. This process has the index "me".

5.3.1 Case A 1: Multidimensional Finite Difference

Assume that, for each process i, we are given D(i), which is a function that returns the set of global view data object indices required to calculate the gradient on dimension i. Also, we are given nabla(i, values), which is a function that returns the gradient on dimension i, given the value of the global view data object elements specified by D(i).

Implicit

```
for time := 0 to T-1 step h do
  GDS_get(grad_data_buf, D(me), ..., gds_grad)
  GDS_get(position_buf, me, ..., gds_position)
  my_grad := nabla(me, my_data)
  position_buf += my_grad
  GDS_fence(gds_position)
  GDS_put(position_buf, me, ..., gds_position)
  GDS_fence(gds_position)
end for
```

Monotonic Implicit

```
for time := 0 to T-1 step h do
  GDS_get(grad_data_buf, D(me), ..., gds_grad)
  GDS_get(position_buf, me, ..., gds_position)
  my_grad := nabla(me, my_data)
  position_buf += my_grad
  GDS_put(position_buf, me, ..., gds_position)
  GDS_version_inc(gds_position, 1, ...)
end for
```

5.3.2 Case A 2: PGAS Matrix Addition

The PGAS program:

```
shared int A[N][N], B[N][N], C[N][N]
...
forall i := 0 to N-1 do
  for j := 0 to N-1 do
    C[i][j] := A[i][j] + B[i][j]
  end for
end forall
```

Is equivalent to the GVR:

```
GDS_gds_create([N,N], GDS_INT, GDS_WORLD, gds_A)
GDS_gds_create([N,N], GDS_INT, GDS_WORLD, gds_B)
GDS_gds_create([N,N], GDS_INT, GDS_WORLD, gds_C)
...
GDS_get(my_A, [me, [0, N-1]], ..., gds_A)
GDS_get(my_B, [me, [0, N-1]], ..., gds_B)
for j := 0 to N-1 do
  my_C[j] := my_A[j] + my_B[j]
end for
GDS_put(my_c, [me, [0, N-1]], ..., gds_C)
GDS_fence(gds_C)
/* free gds_A, gds_B, and gds_C */
...
```

5.3.3 Case B: Iterative Method (SD) on a Sparse Matrix

We iteratively solve Ax = b for x.

> We use two global view data objects: The first, gds x, stores the current approximate solution. The second, gds r, is used to store the residual for the reduction step. Assume there exists a function partition(i), which, given a process i, returns the set of rows of our global view data object that process i will be responsible for updating. It may also copy said rows to storage local to process i. Let A(rows) and b(rows) return the specified rows of A and b, respectively. Let cols(rows) return the set of columns for which at least one entry is nonzero given a set of rows. Let norm1(gds) return the Manhattan norm of gds.

Implicit

```
my_rows := partition(me)
repeat
  GDS_get(resid_data_buf, cols(my_rows), ..., gds_r)
  GDS_get(position_buf, my_rows, ..., gds_x)
  my_r := A(my_rows) * position_buf - b(my_rows)
  my_grad := STEP_SIZE * 2 * transpose(A(my_rows)) * my_r
  position_buf -= my_grad
  GDS_fence(gds_x)
  GDS_put(position_buf, my_rows, ..., gds_x)
  GDS_fence(gds_x)
  GDS_put(my_r, my_rows, ..., gds_r)
  GDS_fence(gds_r)
  norm = norm1(gds_r) / NUM_ROWS
until norm < TOL
```

Monotonic Implicit

```
my_rows := partition(me)
repeat
  GDS_get(resid_data_buf, cols(my_rows), ..., gds_r)
  GDS_get(position_buf, my_rows, ..., gds_x)
  my_r := A(my_rows) * column_buf - b(my_rows)
  my_grad := STEP_SIZE * 2 * transpose(A(my_rows)) * my_r
  position_buf -= my_grad
  GDS_put(position_buf, my_rows, ..., gds_x)
  GDS_version_inc(gds_x, 1, ...)
  GDS_put(my_r, my_rows, ..., gds_r)
  GDS_version_inc(gds_r, 1, ...)
  norm = norm1(gds_r) / NUM_ROWS
until norm < TOL
```

5.3.4 Case C: Parallel Barnes-Hut in 1 dimension

Assume we have a set of particles labeled 0 to N-1.

> In this example, we have four global view data objects: The first, gds m, is a vector of length N, which signifies the mass of each particle. This array is constant, and never needs synchronization, although it does need to be globally available. The second, gds x, is a vector of length N, which signifies the position of each particle. The third, gds tree, is a matrix with a number of rows on the order of NlogN. This array signifies the structure of trees through some unspecified means. The fourth, gds info, has the same number of rows as gds tree, and specifies each node's center of mass, total length, and total mass. We define a function partition(i), which returns the particles for which process i is responsible.

Implicit

```
/*
 * Calculate initial values for:
 *   my_particles, mass_buf, position_buf, tree_buf, gds_m, gds_x, and
 *   gds_tree.
 */
...
```

```
for time := 0 to T-1 step h do
  /*
   * Calculate local interactions of particles in my_particles using values
   * stored in local buffers.
   */
  info_buf := calculate_local_info(my_particles, position_buf, mass_buf,
              tree_buf)
  GDS_put(info_buf, get_local_info_bounds(my_particles), ... , gds_info)
  GDS_fence(gds_info)
  current_node := parent(local_clusters_node)
  while current_node != root do
    GDS_get(info_buf, info_pertaining_to(current_node), ..., gds_info)
    info_buf := calculate_node_info(current_node, info_buf)
    GDS_fence(gds_info)
    GDS_put(info_buf, info_pertaining_to(current_node), ..., gds_info)
    GDS_fence(gds_info)
    current_node := parent(current_node)
  end while
  /*
   * Calculate global interactions of all particles in my_particles using
   * calls to GDS_get(gds_tree), and calls to GDS_get(gds_info).
   */
  ...
  /*
   * Mutate position_buf based on calculated forces.
   */
  ...
  if repartition_is_required() then
    GDS_put(position_buf, my_particles, ..., gds_x)
    GDS_fence(gds_x)
    my_particles := partition(me)
    GDS_get(mass_buf, my_particles, ..., gds_m)
    GDS_get(position_buf, my_particles, ..., gds_x)
    tree_buf := calculate_local_tree(my_particles, position_buf, mass_buf)
    GDS_put(tree_buf, get_local_tree_bounds(my_particles), ..., gds_tree)
    GDS_fence(gds_tree)
    while global_root_node_does_not_exist() do
      GDS_get(tree_buf, current_least_deep_nodes, ..., gds_tree)
      merge_with_neighbor(tree_buf)
      GDS_fence(gds_tree)
      GDS_put(tree_buf, current_least_deep_nodes, ..., gds_tree)
      GDS_fence(gds_tree)
    end while
    GDS_fence(gds_info)
  end if
end for
```

Monotonic Implicit

```
/*
 * Calculate initial values for:
 *   my_particles, mass_buf, position_buf, tree_buf, gds_m, gds_x, and
 *   gds_tree.
 */
 ...
 for time := 0 to T-1; step h do
  /*
```

```
   * Calculate local interactions of particles in my particles using values
   * stored in local buffers.
   */
  info_buf := calculate_local_info(my_particles, position_buf, mass_buf,
              tree_buf)
  GDS_put(info_buf, get_local_info_bounds(my_particles), ..., gds_info)
  GDS_fence(gds_info)
  current_node := parent(local_clusters_node)
  while current_node != root do
    GDS_get(info_buf, info_pertaining_to(current_node), ..., gds_info)
    info_buf := calculate_node_info(current_node, info_buf)
    GDS_put(gds_info, info_pertaining_to(current_node), info_buf)
    GDS_version_inc(gds_info, 1, ...)
    current_node := parent(current_node)
  end while
  /*
   * Calculate global interactions of all particles in my_particles using
   * calls to GDS_get(gds_tree), and calls to GDS_get(gds_info).
   */
  ...
  /*
   * Mutate position_buf based on calculated forces.
   */
  ...
  if repartition_is_required() then
    GDS_put(position_buf, my_particles, ..., gds_x)
    GDS_fence(gds_x)
    my_particles := partition(me)
    GDS_get(mass_buf, my_particles, ..., gds_m)
    GDS_get(position_buf, my_particles, ..., gds_x)
    tree_buf := calculate_local_tree(my_particles, position_buf, mass_buf)
    GDS_put(tree_buf, get_local_tree_bounds(my_particles), ..., gds_tree)
    GDS_fence(gds_tree)
    while global_root_node_does_not_exist() do
      GDS_get(tree_buf, current_least_deep_nodes, ..., gds_tree)
      merge_with_neighbor(tree_buf)
      GDS_put(tree_buf, current_least_deep_nodes, ..., gds_tree)
      GDS_version_inc(gds_tree, 1, ...)
    end while
    GDS_fence(gds_info)
  end if
end for
```

5.3.5 Case D: 1-Dimensional DMC Task-Oriented Implicit

```
x := me * WIDTH/nprocs - WIDTH/2
task_collection_create(GDS_WORLD, tc)
for t := 0 to T-1 step TIMESTEP do
  for i := 0 to N-1 do
    x' := random_number_in_range(-WIDTH/2, WIDTH/2)
    y' := random_number_in_range(0, HEIGHT)
    task_create(t)
    /* Register gdss to which get will be applied */
    t.get_gdss := [gds_psi]
    /* Bounds of gdss specified in previous line */
    t.get_bounds := [segment containing(x')]
    /* Register gdss to which acc will be applied */
```

```
    t.acc_gdss := [gds_sum]
    /* Bounds of gdss specified in previous line */
    t.acc bounds := [me]
    t.func_to_exec := LESS_THAN_INTEGRAND
    t.func_args := (me, x, x', y')
    t.affinity := me
    task_enqueue(t, tc)
  end for
  task_collection_sync(tc)
  /*
   * Either ensures that all tasks currently in the collection are complete,
   * or puts the burden on the scheduler to ensure that the result of the
   * computation is identical to the result that would have occured if all
   * the tasks in the collection had been complete at this point
   */
  GDS_get(sum, me, ... , gds_sum)
  area := (sum / N) * WIDTH * HEIGHT
  GDS_put(gds_psi, me, area)
  GDS_version_inc(gds_psi, 1, ...)
end for
```

With auxiliary function:

```
function LESS_THAN_INTEGRAND([gds_psi], [gds_sum], (me, x, x', y'))
  GDS_get(psi_vals, segment_containing(x'), ..., gds_psi)
  psi_val := interpolate(psi_vals)
  if y' < G(x, x') * psi_val then
    GDS_acc(gds_sum, me, 1, GDS_SUM)
  end if
end function
```

5.4 Case studies: Error Handling Scoping

GVR allows applications to create error recovery handlers. In general, these error handlers (EH),

1. EH's need access to some application state for recovery
2. EH's must be visible to the GVR error handling dispatch to be called

So, the critical question is the relationship of the "scopes" of the error recovery state (ERS) and the error handler (EH).

There are several common solutions:

1. Case 1: define ERS and EH globally, allowing the EH to access the required state. This approach requires changing the visibility in the program, thus changing program structure.

#. Case 2: define variable that hold pointers to the ERS in the global scope (root or dictionary) that an EH can reference and then extract and modify the required recovery state. This approach requires minimum changing of program.

Note that, In GVR, the ERS is typically GDS's, thereby providing access to a full set of versions. It is the responsibility of the programmer to ensure that the ERS stays accessible (not freed explicitly or automatically) and up to date. If the ERS is a GDS, the version system makes this relatively simple.

In ddcMD, the application has a collection of global distributed data structures. The program utilizes GDS global arrays to store data. In simulation procedure, data are captured in versions periodically.

Programmers define a global error handler *recovery_func()* with handles application semantic erros and can be generalized to memory errors. The handler is registered with the global data structures, i.e., the data is globally visible to the error handler. When errors are captured by error checks, the application sends error information to GVR, which therefore invokes the global error handler to restore data. An example of case 1.

```
/* ddcMD example for scoping error handler */

/* Global data structure */
struct state {
  GDS_gds_t _position;
  GDS_gds_t _velocity;
  GDS_gds_t _domain;
  GDS_gds_t _time;
  ...
} State;

/* User defined error handler */
recovery_func(gds, error_descriptor) {
  GDS_get(local_data_structure, gds);
  GDS_resume_global(gds, error_desc);
}

main() {
  /* Create global array data structure */
  GDS_alloc(State._time);
  GDS_alloc(State._position);
  ...

  /* Register the specific error handler */
  GDS_register_global_error_handler(State._position, recovery_func);

  /* Molecular Dynamics Simulation Loop */
  simulation_loop() {
  /* Actual computation work */
  computation();

  /* Error detection */
  error = detection_func();
  /* Error handling */
  if (error) {
    /* Create error descriptor for the error */
    error_descriptor = GDS_create_error_descriptor();
    /* Raise the global error */
    GDS_raise_global_error(gds, error_descriptor);
    continue;
  }
  GDS_put(local_data_structure, State._position);
  ...
  /* Take snapshot of correct states */
  if (snapshot point) {
    GDS_version_inc(State._position);
    ...
  }
}

}
```

An example of case 2,

```
// Pointers to state used for recovery
ERS_global_dictionary {
  void* recovery_state_1;
  void* recovery_state_2;
};

main_user_program(){
  float* my_3d_array [1024][256][128];
  gds* my_versioned_gds = gds_alloc(...);

  register_error_handler(my_error_handler...);

  // Store pointers in global recovery state
  ERS_global_dictionary->recovery_state_1 = my_3d_array;
  ERS_global_dictionary->recovery_state_2 = my_versioned_gds;

  // Do the real Computation
  ...my_versioned_gds...
  ...my_3d_array...
  ...signal an error that leads to error handler invocation...
  ...after handler recovers error, go on with the
  computation...
}

// User-defined error handler
my_error_handler(state) {
  float* temp1 = ERS_global_dictionary->
  recovery_state_1;
  gds* temp2 = ERS_global_dictionary->
  recovery_state_2;
  ...do the recovery using temp1, temp2, state
  ...
}
```

5.5 Case studies: error check, signaling, and recovery

We list two case studies to present the local/global error check, signaling, and recovery.

5.5.1 miniFE

```
function MAIN
  /* create A and b */
  ...
  /* initialize x */
  ...
  GDS_alloc(..., &gds_x)

  /* Define my own error attribute */
  attr_name = "LOCAL_BUFFER_KEY"
  GDS_define_error_attr_key(attr_name, strlen(attr_name),
      GDS_EAVTYPE_INT, &LOCAL_BUFFER_KEY)

  /* Create error matching predicate */
  GDS_create_error_pred(&pred);
```

```
  GDS_create_error_pred_term(LOCAL_BUFFER_KEY,
      GDS_EPRED_ANY, /* Any attribute value will be allowed */
      0, NULL, /* No additional parameter for this predicate */
      &term)
  /* Add a term to the predicate */
  GDS_add_error_pred(pred, term)
  GDS_free_error_pred_term(&term)

  /* Register local error handler */
  GDS_register_local_error_handler(gds_x, pred, reload)

  GDS_free_error_pred(&pred)

  return iter_solve(A, b, x, gds x)
end function

function reload(gds, err_desc) /* Reload the previous version */
  GDS_get_error_attr(err_desc, LOCAL_BUFFER_KEY, local_buffer, ...)
  GDS_descriptor_clone(gds)
  GDS_move_to_prev(gds_copy)
  GDS_get(local_buffer, ..., gds_copy)
  GDS_resume_local(gds)
end function

function iter_solve(A, b, x, gds_x)
  repeat
    old_normr := normr
    r := Ax-b
    normr := norm(r)
    if (old_normr - normr) / old normr > TOL_1 then
      /* Initialize err_desc */
      GDS_create_error_descriptor(&err_desc)
      GDS_add_error_attr(err_desc, LOCAL_BUFFER_KEY, x, sizeof(x))
      /* Trigger the local error handler */
      GDS_raise_local_error(gds_x, err_desc)
      /* err_desc will be freed by GDS_resume_local() */
      do_necessary_recalculation()
    end if
    if iteration % CP_INTERVAL == 0 then
      /* Make a snapshot */
      GDS_put(x, ..., gds_x)
      GDS_version_inc(gds_x, ...)
    end if
    x := do_calculation(A, b, x)
  until normr < TOL_2
  return x
end function
```

5.5.2 miniMD

```
function MAIN
  GDS_alloc(GDS_PRIORITY_HIGH, &gds)
  /* Initialize predicate */
  ...
  /* Register global error handler */
  GDS_register_global_error_handler(gds, pred, rollback)
  /* Register global error check for rollback */
```

```
  GDS_register_global_error_check(gds, full_check)
  return iter_computation()
end function

/* Rollback to the correct version */
function rollback(gds, err_desc)
  GDS_move_to_prev(gds)
  while GDS_check_global_error(gds) != OK do
    GDS_move_to_prev(gds)
  end while
  GDS_get(atoms, gds)
  GDS_resume_global(gds, err_desc)
end function

function iter_computation
  repeat
    do_comp_and_comm(atoms)
    if atoms_out_of_box(atoms) then
      /* Initialize error descriptor */
      GDS_create_error_descriptor(&global_error_desc)
      GDS_add_error_attr(global_error_desc, ...)
      ...
      /* Raised global error will be triggered at the next
         stable point (i.e. GDS_version_inc or GDS_fence) */
      GDS_raise_global_error(gds, global_error_desc)
    end if
    GDS_put(atoms, gds)
    GDS_version_inc(gds)
  until converge()
end function
```

CHAPTER

SIX

THE GVR INTERFACE 1.0.0

6.1 Initialization

GDS_init (*argc, argv, requested_thread_support, provided_thread_support*)

Initializes GDS library. Must be called before any other GDS functions. The *requested_thread_support* argument should be passed — and the *provided_thread_support* argument will return — one of the following, predefined values, which are inherited from MPI.

- GDS_THREAD_SINGLE Each process has exactly one thread.
- GDS_THREAD_FUNNELED Each process may have multiple threads, but only the main thread may call the GVR runtime system functions.
- GDS_THREAD_SERIALIZED Each process may have multiple threads, and any thread may call the GVR runtime system functions. However, the application should arbitrate among the threads to ensure that at most one thread calls the GVR runtime system at the same time.
- GDS_THREAD_MULTIPLE Each process may have multiple threads and any thread may call the GVR runtime system functions. Multiple threads may call the GVR runtime system functions concurrently.

Parameters:

- **argc**: the number of command line arguments (IN::integer)
- **argv**: command line arguments (INOUT::array of string)
- **requested_thread_support**: the thread concurrency level requested by the program (IN::GDS_thread_support_t)
- **provided_thread_support**: actual thread concurrency level provided by the runtime system. Might be lower than the requested level depending on the runtime system's implementation (OUT::GDS_thread_support_t)

C:

```
GDS_status_t GDS_init(int argc, char **argv[],
  GDS_thread_support_t requested_thread_support,
  GDS_thread_support_t *provided_thread_support)
```

Fortran:

```
GDS_INIT(requested_thread_support, provided_thread_support, status)
  INTEGER, INTENT(IN) :: requested_thread_support
  INTEGER, INTENT(OUT) :: provided_thread_support
  INTEGER, INTENT(OUT) :: status
```

GDS_finalize ()

Cleans up GDS library. Must be called after all other GDS procedures have completed.

C:

```
GDS_status_t GDS_finalize()
```

Fortran:

```
GDS_FINALIZE(status)
  INTEGER, INTENT(OUT) :: status
```

GDS_comm_rank (*comm, rank*)

Get the calling process's rank within a given communicator.

Parameters:

- **comm**: the communicator (IN::GDS_comm_t)
- **rank**: the rank of calling process (OUT::integer)

C:

```
GDS_status_t GDS_comm_rank(GDS_comm_t comm, int *rank)
```

Fortran:

```
GDS_COMM_RANK(comm, rank, status)
  INTEGER, INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: rank
  INTEGER, INTENT(OUT) :: status
```

GDS_comm_size (*comm, size*)

Get the size of the given communicator.

Parameters:

- **comm**: the communicator (IN::GDS_comm_t)
- **size**: the size of the communicator (OUT::integer)

C:

```
GDS_status_t GDS_comm_size(GDS_comm_t comm, int *size)
```

Fortran:

```
GDS_COMM_SIZE(comm, size, status)
  INTEGER, INTENT(IN) :: comm
  INTEGER, INTENT(OUT) :: size
  INTEGER, INTENT(OUT) :: status
```

6.2 Creating a Global Data Structure

GDS_create (*ndims, count, element_type, global_to_local_func, local_to_global_func, local_buffer, local_buffer_count, resilience_priority, users, info, gds*)

Creates a GDS object. A collective call that fuses a set of local memory buffers into a GDS object, enabling the use of said buffers as a global data structure by all processes specified in the users argument. A process can supply a null memory buffer.

Parameters:

- **ndims**: the number of dimensions in the global array (IN::GDS_size_t)
- **count**: the number of elements in each dimension of the global array (IN::GDS_size_t)
- **element_type**: type of elements in the GDS (IN::GDS_datatype_t)
- **global_to_local_function**: function mapping global indices to local indices. *global_indices* is an input value. It is an ndims-length array signifying a single location in the global address space. *local_rank* is an output value. It is a scaler that signifies the process rank associated with the given global location. *local_offset* is an output value. It is a scalar that signifies the offset in the buffer provided by the process specified in the rank argument that corresponds with the given global indices. This argument is expressed in terms of elements. The function passed here must return the same results for every process. That is, every process should pass the same function pointer for this argument (IN::GDS_status_t (*global_to_local_func) (GDS_size_t global_indices[], GDS_size_t *local_rank, GDS_size_t *local_offset))
- **local_to_global_function**: function mapping local indices to global indices. local_offset is an input value. It is a scalar signifying an offset in the buffer provided by the calling process. This offset is specified in terms of number of elements. global_indices is an output value. It should output an ndims-length array signifying the global indices corresponding to *local_offset* for the calling process. This function is expected to return different results for every process. The calling process should provide global indices corresponding to its own provided buffer. (IN::GDS_status_t (*local_to_global_func) (GDS_size_t local_offset, GDS_size_t *global_indices))
- **local_buffer**: base address of local buffer (IN::void*)
- **local_buffer_count**: size of the local buffer in terms of the number of elements of the type defined by the *element_type* argument (IN::GDS_size_t)
- **resilience_priority**: desired resilience priority for the GDS (IN::GDS_priority_t)
- **users**: communicator of processes that can access the GDS object (IN::GDS_comm_t)
- **info**: hints. *GDS_info_t* is a direct mapping of *MPI_Info*. See the MPI document for functions to manipulate an *MPI_Info* object. *GDS_create* will recognize *GDS_ORDER_KEY* (IN::GDS_info_t)
- **gds**: returned handle to created GDS (OUT::GDS_gds_t)

C:

```
GDS_status_t GDS_create(GDS_size_t ndims, const GDS_size_t count[],
  GDS_datatype_t element_type,
  (GDS_status_t (*global_to_local_func)
      (const GDS_size_t global_indices[], int *local_rank, GDS_size_t *local_offset)),
  (GDS_status_t (*local_to_global_func)
      (GDS_size_t local_offset, GDS_size_t *global_indices)),
  void* local_buffer, GDS_size_t local_buffer_count,
  GDS_priority_t resilience_priority, GDS_comm_t users,
  GDS_info_t info, GDS_gds_t *gds)
```

Fortran:

```
GDS_CREATE(ndims, count, element_type, &
  global_to_local_func, local_to_global_func, &
  local_buffer, local_buffer_count, &
  resilience_priority, users, info, gds, status)
  INTEGER(8), INTENT(IN) :: ndims
```

```
  INTEGER(8), DIMENSION(*), INTENT(IN) :: count
  INTEGER, INTENT(IN) :: element_type
  TYPE(C_FUNPTR), VALUE, INTENT(IN) :: global_to_local_func
  TYPE(C_FUNPTR), VALUE, INTENT(IN) :: local_to_global_func
  TYPE(C_PTR), VALUE, INTENT(IN) :: local_buffer
  INTEGER(8), INTENT(IN) :: local_buffer_count
  INTEGER, INTENT(IN) :: resilience_priority
  INTEGER, INTENT(IN) :: users
  INTEGER, INTENT(IN) :: info
  TYPE(C_PTR), INTENT(OUT) :: gds
  INTEGER, INTENT(OUT) :: status
```

Function Type for global_to_local_function:

C:

```
GDS_status_t global_to_local_func(const GDS_size_t global_indices[],
  int *local_rank, GDS_size_t *local_offset)
```

Fortran:

```
INTEGER(C_INT) FUNCTION VECTOR_GLOBAL_TO_LOCAL(global_indices, local_rank, &
  local_offset) BIND(C)
  USE, INTRINSIC :: ISO_C_BINDING
  INTEGER(C_SIZE_T), DIMENSION(*), INTENT(IN) :: global_indices
  INTEGER(C_INT), INTENT(OUT) :: local_rank
  INTEGER(C_SIZE_T), INTENT(OUT) :: local_offset
```

Function Type for local_to_global_function:

C:

```
GDS_status_t local_to_global_func(GDS_size_t local_offset,
  GDS_size_t global_indices[])
```

Fortran:

```
INTEGER(C_INT) FUNCTION VECTOR_LOCAL_TO_GLOBAL(local_offset, global_indices) BIND(C)
  USE, INTRINSIC :: ISO_C_BINDING
  INTEGER(C_INT), VALUE, INTENT(IN) :: local_offset
  INTEGER(C_SIZE_T), DIMENSION(*), INTENT(OUT) :: global_indices
```

Predefined Constants for Type: GDS_info_t

- **GDS_ORDER_KEY**: specifies data ordering of a multi-dimensional array. The value should be one of: *GDS_ORDER_DEFAULT, GDS_ORDER_ROW_MAJOR, GDS_ORDER_COL_MAJOR*. *GDS_ORDER_DEFAULT* is translated to language-dependent default: row major ordering in C binding and column major in Fortran binding.

GDS_alloc (*ndims, count, min_chunks, element_type, resilience_priority, users, info, gds*)

A collective call that creates a GDS object that is accessible to all the users. Allocates the needed memory with layout as specified by the arguments.

Parameters:

- **ndims**: the number of dimensions in array (IN::GDS_size_t)
- **count**: the number of elements in each dimension (IN::array with ndims elements of GDS_size_t)

- **min_chunks**: the minimum chunk size in each dimension in number of elements (IN::array with ndims elements of GDS_size_t)
- **element_type**: type of elements in the GDS (IN::GDS_datatype_t)
- **resilience_priority**: desired resilience priority for the gds (IN::GDS_priority_t)
- **users**: communicator of processes that can access the gds object (IN::GDS_comm_t)
- **info**: hints. See the description of the info argument of `GDS_create` for details (IN::GDS_info_t)
- **gds**: returned handle to created gds (OUT::GDS_gds_t)

C:

```
GDS_status_t GDS_alloc(GDS_size_t ndims, const GDS_size_t count, const
  GDS_size_t min_chunks, GDS_datatype_t element_type,
  GDS_priority_t resilience_priority, GDS_comm_t users, GDS_info_t info,
  GDS_gds_t gds)
```

Fortran:

```
GDS_ALLOC(ndims, count, min_chunks, element_type, &
  resilience_priority, users, info, gds, status)
  INTEGER(8), INTENT(IN) :: ndims
  INTEGER(8), DIMENSION(*), INTENT(IN) :: count
  INTEGER(8), DIMENSION(*), INTENT(IN) :: min_chunks
  INTEGER, INTENT(IN) :: element_type
  INTEGER, INTENT(IN) :: resilience_priority
  INTEGER, INTENT(IN) :: users
  INTEGER, INTENT(IN) :: info
  TYPE(C_PTR), INTENT(OUT) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_free (*gds*)

Frees the GDS object gds and returns a null handle (equal to *GDS_NULL*). This is a collective call executed by all of the processes that can access the gds. If created with *GDS_alloc*, *GDS_free* will free corresponding allocated memory.

Parameters:

- **gds**: gds handle to free (INOUT:: GDS_gds_t)

C:

```
GDS_status_t GDS_free(GDS_gds_t *gds)
```

Fortran:

```
GDS_FREE(gds, status)
  TYPE(C_PTR), INTENT(OUT) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_get_attr (*gds, attribute_key, attribute_value, flag*)

Queries attributes for the given GDS object. If the specified attribute is present, the flag argument is set to true. Returns an error if an invalid attribute key is specified. The following attributes are predefined and are required to be present:

- GDS_TYPE
- GDS_CREATE_FLAVOR

- GDS_GLOBAL_LAYOUT
- GDS_NUMBER_DIMENSIONS
- GDS_COUNT
- GDS_CHUNK_SIZE

Parameters:

- **gds**: GDS object to query (IN::GDS_gds_t)
- **attribute_key**: Attribute key (IN::GDS_attr_t)
- **attribute_value**: attribute value (OUT::arbitrary)
- **flag**: boolean indicating if the attribute is present (OUT::boolean)

C:

```
GDS_status_t GDS_get_attr(GDS_gds_t gds, GDS_attr_t attribute_key,
  void *attribute_value, int *flag)
```

Fortran:

```
GDS_GET_ATTR(gds, attr_key, attr_val, flag, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, VALUE, INTENT(IN) :: attr_key
  TYPE(C_PTR), VALUE, INTENT(IN) :: attr_val
  INTEGER, INTENT(OUT) :: flag
  INTEGER, INTENT(OUT) :: status
```

Predefined Constants for Type: GDS_attr_t

- **GDS_TYPE**: the datatype of the GDS object. (GDS_datatype_t)
- **GDS_CREATE_FLAVOR**: indicates how the GDS object was created. One of: GDS_FLAVOR_CREATE, GDS_FLAVOR_ALLOC (GDS_flavor_t)
- **GDS_GLOBAL_LAYOUT**: the global layout of the GDS. Not supported yet.
- **GDS_NUMBER_DIMENSIONS**: a GDS_size_t signifying the number of dimensions of the GDS. (GDS_size_t)
- **GDS_COUNT**: an array of GDS_size_t containing a number of elements equal to the number of dimensions of the GDS. Signifies the size of every dimension in the GDS in terms of number of elements. User should be responsible to provide array buffer with sufficient length. (GDS_size_t [])
- **GDS_CHUNK_SIZE**: an array of GDS_size_t containing a number of elements equal to the number of dimensions of the GDS. Signifies the size of every dimension in a single chunk of the GDS in terms of number of elements. Defined only if GDS was created with GDS_alloc. User should be responsible to provide array buffer with sufficient length. (GDS_size_t [])

GDS_get_comm (*gds, comm*) Returns a duplicate of the communicator used to create the GDS object gds.

Parameters:

- **gds**: GDS object (IN::GDS_gds_t)
- **comm**: GDS communicator (OUT::GDS_comm_t)

C:

```
GDS_status_t GDS_get_comm(GDS_gds_t gds, GDS_comm_t *comm)
```

Fortran:

```
GDS_GET_COMM(gds, comm, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: comm
  INTEGER, INTENT(OUT) :: status
```

6.3 Using a Global Data Structure

GDS_put (*origin_addr, origin_ld, lo_index, hi_index, gds*)

Puts data in the GDS memory. Transfers a block of entries from the origin, starting at origin_addr, to the segment of the GDS object specified by `lo_index` and `hi_index`. Multiple `GDS_put` operations to the same gds memory location can lead to undefined output at the target location. Further, there is no ordering of GDS_put operations whatsoever, unless additional synchronization is used to explicitly order them. In general, put returns immediately — it is nonblocking. Contents of the buffer pointed by *origin_addr* should not be modified until the operation is completed by a synchronization function (e.g. *GDS_wait*). This function returns an error if applied to a version of the gds that is not the current version.

Parameters:

- **origin_addr**: address of the local buffer from which data will be copied to the gds (IN::void*)
- **origin_ld**: defines the shape of local buffer in units of the element_type (IN::array with ndims-1 elements of GDS_size_t)
- **lo_index**: starting element of remote buffer (IN::array with ndims elements of GDS_size_t)
- **hi_index**: ending element of remote buffer (IN::array with ndims elements of GDS_size_t)
- **gds**: the gds in which data will be put (IN::GDS_gds_t)

C:

```
GDS_status_t GDS_put(void *origin_addr, GDS_size_t origin_ld[],
  GDS_size_t lo_index[], GDS_size_t hi_index[], GDS_gds_t gds);
```

Fortran:

```
GDS_PUT(orig_addr, orig_ld, lo_idx, hi_idx, gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: orig_addr
  INTEGER(8), DIMENSION(*), INTENT(IN) :: orig_ld
  INTEGER(8), DIMENSION(*), INTENT(IN) :: lo_idx
  INTEGER(8), DIMENSION(*), INTENT(IN) :: hi_idx
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_get (*origin_addr, origin_ld, lo_index, hi_index, gds*)

Gets data from the GDS memory. Similar to *GDS_put*, except that the direction of data transfer is reversed. Data is copied from the target memory to the origin. The copied data must fit,

without truncation, in the origin buffer. In general, get returns immediately — it is nonblocking. A synchronization function *GDS_wait*, *GDS_wait_local* or *GDS_fence* should be used before the buffer is considered valid.

Parameters:

- **origin_addr**: address of the local buffer to which data will be copied from the gds (IN::void*)
- **origin_ld**: defines the shape of local buffer in units of the element type (IN::array with ndims-1 elements of GDS_size_t)
- **lo_index**: starting element of remote buffer (IN::array with ndims elements of GDS_size_t)
- **hi_index**: ending element of remote buffer (IN::array with ndims elements of GDS_size_t)
- **gds**: the gds from which data will be copied (IN::GDS_gds_t)

C:

```
GDS_status_t GDS_get(void *origin_addr, GDS_size_t origin_ld[],
  GDS_size_t lo_index[], GDS_size_t hi_index[], GDS_gds_t gds);
```

Fortran:

```
GDS_GET(orig_addr, orig_ld, lo_idx, hi_idx, gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: orig_addr
  INTEGER(8), DIMENSION(*), INTENT(IN) :: orig_ld
  INTEGER(8), DIMENSION(*), INTENT(IN) :: lo_idx
  INTEGER(8), DIMENSION(*), INTENT(IN) :: hi_idx
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_acc (*origin_addr, origin_ld, lo_index, hi_index, accumulate_op, gds*)

Accumulates the contents of the origin buffer (as defined by *origin_addr* and *origin_ld*) to the specified segment of the GDS object, using the operation op. This is like *GDS_put* except that data are combined into the target area instead of the data in the target area being overwritten. This function is a non-blocking operation.

Parameters:

- **origin_addr**: address of the local buffer to which data will be accumulated to the gds (IN::void*)
- **origin_ld**: defines the shape of local buffer in units of the element type (IN::array with ndims-1 elements of GDS_size_t)
- **lo_index**: starting element of remote buffer (IN::array with ndims elements of GDS_size_t)
- **hi_index**: ending element of remote buffer (IN::array with ndims elements of GDS_size_t)
- **accumulate_op**: GDS accumulate operation used to accumulate date into the gds object (IN::GDS_op_t)
- **gds**: the gds from which data will be copied (IN::GDS_gds_t)

Returns: an error if applied to a version of the GDS object that is not the current version.

C:

```
GDS_status_t GDS_acc(void *origin_addr, GDS_size_t origin_ld[],
  GDS_size_t lo_index[], GDS_size_t hi_index[],
  GDS_op_t accumulate_op, GDS_gds_t gds);
```

Fortran:

```
GDS_ACC(orig_addr, orig_ld, lo_idx, hi_idx, acc_op, gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: orig_addr
  INTEGER(8), DIMENSION(*), INTENT(IN) :: orig_ld
  INTEGER(8), DIMENSION(*), INTENT(IN) :: lo_idx
  INTEGER(8), INTENT(IN) :: hi_idx
  INTEGER, VALUE, INTENT(IN) :: acc_op
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_get_acc (*origin_addr, origin_ld, result_addr, result_ld, lo_index, hi_index, accumulate_op, gds*)

Gets data from GDS memory and performs an operation on the target. Accumulates elements from the origin buffer *origin_addr* to the specified region of the GDS object using the operation op, and returns, in the result buffer *result_addr*, the content of the target region of the gds before the accumulation. The origin and result buffers (*origin_addr* and *result_addr*) must be disjoint. The result of the get operation must fit in the memory region pointed to by *result_addr*. This function is a non-blocking operation.

Parameters:

- **origin_addr**: address of the local buffer to which data will be accumulated to the gds (IN::void*)
- **origin_ld**: defines the shape of local buffer in units of the element type (IN::void*)
- **result_addr**: address of the local buffer to which current state of the gds region is written (IN::void*)
- **result_ld**: shape of local result buffer (IN::array with ndims-1 elements of GDS_size_t)
- **lo_index**: starting element of remote buffer (IN::array with ndims elements of GDS_size_t)
- **hi_index**: ending element of remote buffer (IN::array with ndims elements of GDS_size_t)
- **accumulate_op**: GDS accumulate operation used to accumulate date into the gds object (IN::GDS_op_t)
- **gds**: the gds from which data will be copied (IN::GDS_gds_t)

Returns: an error if applied to a version of the GDS object that is not the current version.

C:

```
GDS_status_t GDS_get_acc(void *origin_addr, GDS_size_t origin_ld[],
  void *result_addr, GDS_size_t result_ld[],
  GDS_size_t lo_index[], GDS_size_t hi_index[],
  GDS_op_t accumulate_op, GDS_gds_t gds);
```

Fortran:

```
SUBROUTINE GDS_GET_ACC(origin_addr, origin_ld, result_addr, ressult_ld,
  lo_index, hi_index, accumulate_op, gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: origin_addr
  INTEGER(8), DIMENSION(*), INTENT(IN) :: origin_ld
```

```
  TYPE(C_PTR), VALUE, INTENT(IN) :: result_addr
  INTEGER(8), DIMENSION(*), INTENT(IN) :: result_ld
  INTEGER(8), DIMENSION(*), INTENT(IN) :: lo_index
  INTEGER(8), DIMENSION(*), INTENT(IN) :: hi_index
  INTEGER, VALUE, INTENT(IN) :: accumulate_op
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_compare_and_swap (*compare_addr, swap_source_addr, swap_result_addr, gds_offset, gds*)

Performs a compare and swap operation. This function compares one element in the compare buffer *compare_addr* with the target element of the GDS *gds_offset* and replaces the value at the target with the value in the origin buffer *swap_source_addr* if the compare buffer and the target element in the target gds are identical. The original value at the target gds is returned in the buffer *swap_result_addr*. All local buffers must be disjoint. This function is a non-blocking operation.

Parameters:

- **compare_addr**: address of the buffer which contains an element which will be compared to the corresponding element in the target gds (IN::void*)
- **swap_source_addr**: address of the buffer which contains an element which may be written into the corresponding element in the target gds (IN::void*)
- **swap_result_addr**: address of the buffer to which the previous value of the target gds will be written (IN::void*)
- **gds_offset**: index of the target element in the target gds (IN::array with ndims elements of GDS_size_t)
- **gds**: the target GDS object (IN::GDS_gds_t)

C:

```
GDS_status_t GDS_compare_and_swap(void *compare_addr, void *swap_source_addr,
  void *swap_result_addr, GDS_size_t gds_offset[], GDS_gds_t gds);
```

Fortran:

```
GDS_COMPARE_AND_SWAP(orig_addr, orig_ld, lo_idx, hi_idx, acc_op, gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: compare_addr
  TYPE(C_PTR), VALUE, INTENT(IN) :: swap_source_addr
  TYPE(C_PTR), VALUE, INTENT(IN) :: swap_result_addr
  INTEGER(8), DIMENSION(*), INTENT(IN) :: gds_offset
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

Notes Due to the limitation of MPI, The datatype of GDS array using this function must belong to one of the following categories of predefined datatypes: C integer, Fortran integer, Logical, Multi-language types or Byte.

GDS_access (*gds, lo_index, hi_index, buffer_type, access_buffer, access_handle*)

Depending on the runtime, the environment, and the value of the buffer_type argument, this function will return a pointer to either:

- the buffer that contains the portion of the current version of the GDS that is stored locally, or,
- a buffer that contains a duplicate of the data that is contained in the portion of the current version of the GDS that is stored locally.

All operations on the specified GDS region are blocked until *gds_release* is called on *access_handle*.

Parameters:

- **gds**: the GDS object (IN::GDS_gds_t)
- **lo_index**: the index of the first element from which the buffer will acquire its data (IN::array with ndims elements of *GDS_size_t*)
- **hi_index**: the index of the last element from which the buffer will acquire its data (IN::array with ndims elements of *GDS_size_t*)
- **buffer_type**: specifies the requested nature of the returned buffer. The value passed should be one of: *GDS_ACCESS_BUFFER_DIRECT* if the pointer should point directly to the buffer used by the GDS to store the local portion of its data, or, *GDS_ACCESS_BUFFER_COPY* if the pointer should point to a buffer containing a copy of the portion of the GDS stored locally, or, *GDS_ACCESS_BUFFER_ANY* to let the runtime decide between the above choices.
- **access_buffer**: buffer pointing to the address specified in the buffer_type argument (OUT::void*)
- **access_handle**: handle to track this GDS_access operation. This will be a required argument for GDS_release (OUT::GDS_access_handle_t)

Notes If the array is created by *GDS_alloc*, then *lo_index* and *hi_index* is checked against the actual global array range. In case of check failure, the return value is *GDS_STATUS_RANGE*, indicating an error. For array created by *GDS_create*, this function returns the buffer provided by user during the call of *GDS_create*.

C:

```
GDS_status_t GDS_access(GDS_gds_t *gds, GDS_size_t lo_index[],
  GDS_size_t hi_index[], GDS_access_buffer_t buffer_type,
  void *access_buffer, GDS_access_handle_t *access_handle)
```

Fortran:

```
GDS_ACCESS(gds, lo_index, hi_index, buf_type, access_buf, access_handle, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER(8), DIMENSION(*), INTENT(IN) :: lo_index
  INTEGER(8), DIMENSION(*), INTENT(IN) :: hi_index
  INTEGER, VALUE, INTENT(IN) :: buf_type
  TYPE(C_PTR), INTENT(OUT) :: access_buf
  TYPE(C_PTR), INTENT(OUT) :: access_handle
  INTEGER, INTENT(OUT) :: status
```

GDS_get_access_buffer_type (*access_handle, buffer_type*)

Returns a value signifying the nature of the buffer returned by *GDS_access*.

Parameters:

- **access_handle**: handle for the access operation that should be queried (IN::GDS_access_handle_t)
- **buffer_type**: the nature of the buffer returned in the call to access. Should be one of: *GDS_ACCESS_BUFFER_DIRECT* or *GDS_ACCESS_BUFFER_COPY* (OUT::GDS_access_buffer_t)

C:

```
GDS_status_t GDS_get_access_buffer_type(GDS_access_handle_t access_handle,
  GDS_access_buffer_t *buffer_type);
```

Fortran:

```
GDS_GET_ACCESS_BUFFER_TYPE(access_handle, buffer_type, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: access_handle
  INTEGER, INTENT(OUT) :: buffer_type
  INTEGER, INTENT(OUT) :: status
```

GDS_release (*access_handle*)

Puts the changes made to the contents of the access_buffer into the gds, and ensure that all subsequent read operations will see the new data.

Parameters:

- **access_handle**: handle that was output by a particular call to GDS_access (IN-OUT::GDS_access_handle_t)

C:

```
GDS_status_t GDS_release(GDS_access_handle_t access_handle);
```

Fortran::

GDS_RELEASE(access_handle, status) TYPE(C_PTR), VALUE, INTENT(IN) :: access_handle INTEGER, INTENT(OUT) :: status

GDS_fence (*gds*)

A collective operation for synchronization. All read operations following the fence will reflect all of the write operations preceding it. If *GDS_ROOT* is specified, this function performs fence operation for all gds arrays.

Parameters:

- **gds**: the GDS object on which the fence takes place (IN::GDS_gds_t)

C:

```
GDS_status_t GDS_fence(GDS_gds_t *gds);
```

Fortran:

```
GDS_FENCE(gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_wait (*gds*)

Blocks until all operations on the gds are completed. The return of a call to GDS_wait signifies that, for the given gds, get, get_acc, and compare_and_swap (all operations returning values) have completed and their values are in the local buffers. In addition, put operations, and other one-sided operations, are completed locally.

Parameters:

- **gds**: the GDS object on which the wait takes place (IN::GDS_gds_t)

C:

```
GDS_status_t GDS_wait(GDS_gds_t *gds);
```

Fortran:

```
GDS_WAIT(gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_wait_local (*gds*)

Blocks until all operations on the gds that impact data stored on the calling process are completed. Similar to GDS_wait, except only waits on reads from, and writes to, buffers local to the calling process.

Parameters:

- **gds**: the GDS object on which the wait takes place (IN::GDS_gds_t)

C:

```
GDS_status_t GDS_wait_local(GDS_gds_t *gds)
```

Fortran:

```
GDS_WAIT_LOCAL(gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

6.4 Versioning, Error-Signaling, Error-Checking, and Error-Recovery

6.4.1 Demarcating Versions

GDS_version_inc (*gds, increment, label, label_size*)

Advances the version for the given GDS object. The runtime may choose whether or not to actually create a new version of the GDS object.

Parameters:

- **gds**: the GDS object (IN::GDS_gds_t)
- **increment**: the number of versions by which the current version number is incremented (IN::GDS_size_t)
- **label**: additional information that should be associated with the version to which the GDS object is advanced. Labels need not be unique (IN::string)
- **label_size**: the size of version label (IN::size_t)

C:

```
GDS_status_t GDS_version_inc(GDS_gds_t gds, GDS_size_t increment,
  const char *label, size_t label_size);
```

Fortran:

```
GDS_VERSION_INC(gds, incre, label, label_size, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER(8), VALUE, INTENT(IN) :: incre
  CHARACTER, DIMENSION(*), INTENT(IN) :: label
  INTEGER(8), VALUE, INTENT(IN) :: label_size
  INTEGER, INTENT(OUT) :: status
```

6.4.2 Error Signaling, Checking and Recovery

Terminologies

Local only one process is involved.

Global all the processes using GDS object are involved.

Stable point a specific point when no process accesses the data via GDS operations and the array contents are in consistent status.

Error Attributes

In GVR, errors are described with *error attributes*. Each attribute is a pair of *error attribute key* and *error attribute value*, which gives a parameter to describe the details of the error. For example, *GDS_EATTR_MEMORY_OFFSET* and *GDS_EATTR_MEMORY_COUNT* are two error attribute keys which describe damaged memory region. An attribute value has a type, specified by *GDS_error_attr_value_type_t*, so that all the components around errors can agree on the interpretation of the value data.

Predefined Constants for Type: GDS_error_attr_value_type_t

- **GDS_EAVTYPE_BYTE**: single 1-byte integer
- **GDS_EAVTYPE_INT**: single 64-bit integer
- **GDS_EAVTYPE_FLOAT**: single 64-bit floating point value
- **GDS_EAVTYPE_BYTE_ARRAY**: an array of 1-byte integer
- **GDS_EAVTYPE_INT_ARRAY**: an array of 64-bit integer
- **GDS_EAVTYPE_FLOAT_ARRAY**: an array of 64-bit floating point value
- **GDS_EAVTYPE_MPIOBJ**: single handle to an MPI object
- **GDS_EAVTYPE_MPIOBJ_ARRAY**: an array of MPI objects

A set of error attributes form an *error descriptor*. In the event of an error, an error descriptor is generated and passed across components which are involved in the error handling path.

Pre-defined Error Attributes

GVR predefines the following error attributes. Type for each attribute is one of *GDS_error_attr_value_type_t*. The prefix *GDS_EAVTYPE_* is omitted for conciseness.] **Error Attributes Related to Data Corruption Errors**

- **GDS_EATTR_MEMORY_VADDR**: starting virtual address of an affected region (INT)
- **GDS_EATTR_MEMORY_SIZE**: size of affected region in bytes (INT)
- **GDS_EATTR_MEMORY_AFFECTED_SIZE**: size of the region actually affected by an error in bytes(INT)
- **GDS_EATTR_MEMORY_N_RANGES**: number of array index ranges affected by an error (INT)
- **GDS_EATTR_MEMORY_DATA**: remaining (possibly corrupted) data (BYTE_ARRAY)
- **GDS_EATTR_MEMORY_CHECK_BITS**: check bits information regarding corrupted region, provided by hardware or other lower component (BYTE_ARRAY)

- **GDS_EATTR_MEMORY_LOCATION_REUSABLE**: if nonzero, memory is corrupted, but new data can be written into the same location. If zero, data cannot be written into the corrupted location again (INT)
- **GDS_EATTR_APP**: attribute to signify an application-semantic error. Value has meaning only to the application (BYTE_ARRAY)
- **GDS_EATTR_DETECTED_BY**: name or other identification of component that detected the error (BYTE_ARRAY)
- **GDS_EATTR_GDS_INDEX**: start index of an affected region in a GDS (INT_ARRAY)
- **GDS_EATTR_GDS_COUNT**: size and shape of affected region in number of elements (INT_ARRAY)
- **GDS_EATTR_GDS_AFFECTED_COUNT**: size and shape of the region actually affected by an error in number of elements (INT_ARRAY)
- **GDS_EATTR_GDS_VERSION**: version number of the corrupted version (INT)
- **GDS_EATTR_GDS_LATENCY**: latency between actual data corruption and detection (INT)

Error Attributes Related to Resource Loss Errors

- **GDS_EATTR_LOST_PROCESSES**: an *MPI_Group* object which describes lost processes (MPIOBJ)
- **GDS_EATTR_LOST_COMMUNICATOR**: an *MPI_Comm* object which describes the old communcator before an error happens (MPIOBJ)
- **GDS_EATTR_REPLACED_COMMUNICATOR**: an *MPI_Comm* object which describes the new communicator after an error happens (MPIOBJ)
- **GDS_EATTR_LOST_COMPUTATION**: amount of lost computation (FLOAT)
- **GDS_EATTR_MPI_RANK_UNRESPONSIVE**: specifies an mpi rank that is not responding to communication (INT)

Error Handlers

Applications can define their own error handlers to deal with errors. Each error handler is associated with an *error predicate* to describe what kind of error attributes can be handled. Error predicate is a set of *terms*, which is a tuple of (attribute key, match expression type, value(s)...).

Match expression type must be one of the followings.

Predefined Constants for Type: GDS_error_match_expr_t

- **GDS_EMEXP_ANY**: matches if the specified attribute key appears in the error descriptor, regardless of its value.
- **GDS_EMEXP_VALUE**: matches if the specified attribute key appears in the error descriptor, and its value equals to the specified value
- **GDS_EMEXP_RANGE**: matches if the specified attribute key appears in the error descriptor, and its value fits within a certain value range. A user specifies (min, max) values and it matches if an error descriptor has a value v which satisfies min <= v <= max. If a value is an array, the term matches if an error descriptor has a value v[N] which satisfies min[i] <= v[i] <= max[i] for all i < N.
- **GDS_EMEXP_NOT**: matches if the specified attribute key does not appear in the error descriptor.

Applications can register multiple error handlers. When an error is raised, GVR invokes one error handler all of whose error matching predicates are satisfied. If there are more than one such an error handler, GVR chooses one according to the following rule.

1. A handler with the biggest number of predicate terms will be chosen
2. If there are still multiple candidates with the above rule, pick one with the biggest number of value matching (*GDS_EMEXP_VALUE*)
3. If there are still multiple candidates with the above rules, pick one with the biggest number of range matching (*GDS_EMEXP_RANGE*).
4. Further rules may be added in future, but currently it is erroneous to have multiple error handlers which cannot be narrowed down to a single handler with above rules.

API Functions

GDS_define_error_attr_key (*attr_name, attr_name_size, value_type, new_key*)

Defines a new error attribute key. This function is a collective call across the entire process in the program, and the parameters should be consistent across different processes.

Parameters:

- **attr_name**: name of the new attribute key. (IN::string)
- **value_type**: attribute value type associated with (IN::GDS_error_attr_value_type_t)
- **new_key**: the newly defined attribute key type (OUT::GDS_error_attr_key_t)

C:

```
GDS_status_t GDS_define_error_attr_key(const char *attr_name, GDS_size_t attr_name_size,
  GDS_error_attr_value_type_t value_type, GDS_error_attr_key_t *new_key);
```

Fortran:

```
GDS_DEFINE_ERROR_ATTR_KEY(attr_name, attr_name_size, value_type, new_key, status)
  CHARACTER, DIMENSION(*), INTENT(IN) :: attr_name
  INTEGER(8), VALUE, INTENT(IN) :: attr_name_size
  INTEGER, VALUE, INTENT(IN) :: value_type
  INTEGER, INTENT(OUT) :: new_key
  INTEGER, INTENT(OUT) :: status
```

GDS_create_error_pred (*pred*)

Creates a new error matching predicate object.

Parameters

- **pred**: a newly created error matching predicate object (OUT::GDS_error_pred_t)

C:

```
GDS_status_t GDS_create_error_pred(GDS_error_pred_t *pred);
```

Fortran:

```
GDS_CREATE_ERROR_PRED(pred, status)
  TYPE(C_PTR), INTENT(OUT) :: pred
  INTEGER, INTENT(OUT) :: status
```

GDS_free_error_pred (*pred*)

Frees a new error matching predicate object.

Parameters

- **pred**: a error matching predicate object to be freed (INOUT::GDS_error_pred_t)

C:

```
GDS_status_t GDS_free_error_pred(GDS_error_pred_t *pred);
```

Fortran:

```
GDS_FREE_ERROR_PRED(pred, status)
  TYPE(C_PTR), INTENT(OUT) :: pred
  INTEGER, INTENT(OUT) :: status
```

GDS_create_error_pred_term (*attr_key, match_expr, value_len, values, term*)

Creates a new error matching predicate term object.

Parameters

- **attr_key**: An error attribute key associated with an error matching expression represented in this term (IN::GDS_error_attr_key_t)
- **match_expr**: An error matching expression type (IN::GDS_error_match_expr_t)
- **value_len**: Length of value (in bytes) associated with an error matching expression represented in this term (IN::GDS_size_t)
- **values**: A pointer to a memory buffer which contains one or more values associated with an error matching expression represented in this term (IN::void *).
- **term**: a newly created error matching predicate term object (OUT::GDS_error_pred_term_t)

Notes on Values *value_len* and *values* parameters will be used only if *match_expr* equals to *GDS_EMEXP_VALUE* or *GDS_EMEXP_RANGE*. When specifying ranges (*GDS_EMEXP_RANGE*), *values* should point to a memory buffer which contains two elements, minimum (first element) and maximum (second element). *value_len* shall be the total size of two elements. When specifying array range, the first half of the buffer is for the minimum and the second half is for the maximum.

C:

```
GDS_status_t GDS_create_error_pred_term(GDS_error_attr_key_t attr_key,
  GDS_error_match_expr_t match_expr, GDS_size_t value_len, const void *values,
  GDS_error_pred_term_t *term);
```

Fortran:

```
GDS_CREATE_ERROR_PRED_TERM(attr_key, match_expr, value_len,
  values, term, status)
  INTEGER, VALUE, INTENT(IN) :: attr_key
  INTEGER, VALUE, INTENT(IN) :: match_expr
  INTEGER(8), VALUE, INTENT(IN) :: value_len
  TYPE(C_PTR), VALUE, INTENT(IN) :: values
  TYPE(C_PTR), INTENT(OUT) :: term
  INTEGER, INTENT(OUT) :: status
```

GDS_free_error_pred_term (*term*)

Frees a new error matching predicate term object.

Parameters

- **pred**: a error matching predicate term object to be freed (IN-OUT::GDS_error_pred_term_t)

C:

```
GDS_status_t GDS_free_error_pred_term(GDS_error_pred_t *pred);
```

Fortran:

```
GDS_FREE_ERROR_PRED_TERM(term, status)
  TYPE(C_PTR), INTENT(OUT) :: term
  INTEGER, INTENT(OUT) :: status
```

GDS_add_error_pred_term (*pred, term*)

Adds an error matching predicate term to the predicate.

Parameters

- **pred**: An error matching predicate object (IN::GDS_error_pred_t)
- **term**: An error matching predicate term object to be added to *pred* (IN::GDS_error_pred_term_t)

Notes It is safe to free the *term* object by calling *GDS_free_error_pred_term* when this function returns.

C:

```
GDS_status_t GDS_add_error_pred_term(GDS_error_pred_t pred, const GDS_error_pred_term_t term);
```

Fortran:

```
GDS_ADD_ERROR_PRED_TERM(pred, term, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: pred
  TYPE(C_PTR), VALUE, INTENT(IN) :: term
  INTEGER, INTENT(OUT) :: status
```

GDS_create_error_descriptor (*error_desc*)

Allocates a new empty error descriptor. The returned error descriptor should eventually be passed to either one of *GDS_raise_local_error* or *GDS_raise_global_error*.

Parameters:

- **error_desc**: returned handle to created error descriptor (OUT::GDS_error_t)

C:

```
GDS_status_t GDS_create_error_descriptor(GDS_error_t *error_desc)
```

Fortran:

```
GDS_CREATE_ERROR_DESCRIPTOR(error_desc, status)
  TYPE(C_PTR), INTENT(OUT) :: error_desc
  INTEGER, INTENT(OUT) :: status
```

GDS_free_error_descriptor (*error_desc*)

Frees an error descriptor. Please note that this function is only for cleaning up purpose, for example error descriptor initialization is aborted for some reason. Usually an error descriptor will be passed to either one of *GDS_raise_local_error* or *GDS_raise_global_error*, and will be

freed at the end of an error handling path (by either *GDS_resume_local* or *GDS_resume_global*), so users do not have to free an error descriptor by *GDS_free_error_descriptor*.

Parameters:

- **error_desc**: error descriptor to be freed (INOUT::GDS_error_t)

C:

```
GDS_status_t GDS_free_error_descriptor(GDS_error_t *error_desc)
```

Fortran:

```
GDS_FREE_ERROR_DESCRIPTOR(error_desc, status)
  TYPE(C_PTR), INTENT(OUT) :: error_desc
  INTEGER, INTENT(OUT) :: status
```

GDS_add_error_attr (*error_desc, attr_key, attr_len, attr_val*)

Adds an error attribute to error descriptor object.

Parameters:

- **error_desc**: the error descriptor (IN::GDS_error_t)
- **attr_key**: attribute key to be added (IN::GDS_error_attr_key_t)
- **attr_len**: size of attribute value in bytes (IN::GDS_size_t)
- **attr_val**: attribute value (IN::arbitrary)

C:

```
GDS_status_t GDS_add_error_attr(GDS_error_t error_desc,
  GDS_error_attr_t attr_key, GDS_size_t attr_len, const void *attr_val);
```

Fortran:

```
GDS_ADD_ERROR_ATTR(err_desc, attr_key, attr_len, attr_val, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: err_desc
  INTEGER, VALUE, INTENT(IN) :: attr_key
  INTEGER(8), VALUE, INTENT(IN) :: attr_len
  TYPE(C_PTR), VALUE, INTENT(IN) :: attr_val
  INTEGER, INTENT(OUT) :: status
```

GDS_get_error_attr (*error_desc, attribute_key, attribute_value, flag*)

Queries attributes of the given error descriptor object. If the specified attribute is present, the flag argument is set to true.

Parameters:

- **error_desc**: the error descriptor (IN::GDS_error_t)
- **attribute_key**: attribute key (IN::GDS_error_attr_t)
- **attribute_value**: attribute value (OUT::arbitrary)
- **flag**: boolean indicating if the attribute is present (OUT::boolean)

C:

```
GDS_status_t GDS_get_error_attr(GDS_error_t error_desc,
  GDS_error_attr_t attr_key, void *attr_val, int *flag);
```

Fortran:

```
GDS_GET_ERROR_ATTR(err_desc, attr_key, attr_val, flag, status)
  INTEGER, VALUE, INTENT(IN) :: err_desc
  INTEGER, INTENT(IN) :: attr_key
  TYPE(C_PTR), VALUE, INTENT(IN) :: attr_val
  INTEGER, INTENT(OUT) :: flag
  INTEGER, INTENT(OUT) :: status
```

GDS_get_error_attr_len (*error_desc, attribute_key, attribute_len, flag*)

Queries attribute value length of the given error descriptor object. If the specified attribute is present, the flag argument is set to true.

Parameters:

- **error_desc**: the error descriptor (IN::GDS_error_t)
- **attribute_key**: attribute key (IN::GDS_error_attr_t)
- **attribute_len**: attribute value length (OUT::GDS_size_t)
- **flag**: boolean indicating if the attribute is present (OUT::boolean)

C:

```
GDS_status_t GDS_get_error_attr_len(GDS_error_t error_desc,
  GDS_error_attr_t attr_key, GDS_size_t *attr_len, int *flag);
```

Fortran:

```
GDS_GET_ERROR_ATTR_LEN(err_desc, attr_key, attr_len, flag, status)
  INTEGER, VALUE, INTENT(IN) :: err_desc
  INTEGER, INTENT(IN) :: attr_key
  INTEGER(8), INTENT(OUT) :: attr_len
  INTEGER, INTENT(OUT) :: flag
  INTEGER, INTENT(OUT) :: status
```

GDS_register_local_error_check (*gds, error_check_function, check_priority*)

User programs can implement whatever error checking they would like, and then signal errors to the GVR system using raise_error. The *register_local_error_check()* function allows users to register a local checking function to the calling process. It is assumed these checking routines are uncoordinated with other processes. It will be triggered and run by the GVR system for the registered process. It is assumed these checking routines make appropriate calls to *raise_error()* when they find them. During an error-check, the application must not write to the passed GDS.

Parameters:

- **gds**: GDS on which this error checking routine is being registered (IN::GDS_gds_t)
- **error_check_function**: the checking function, which takes a GDS, the priority of the check. The function is expected to check the given GDS for faults. If a fault is located, this function should call GDS_raise_error with the appropriate arguments (IN::GDS_status_t (*check_func) (GDS_gds_t gds, GDS_priority_t check_priority))
- **check_priority**: desired priority for the error check. High priority means run first (low cost, finds common errors). Medium and low priorities mean run as last resort (may be high cost, finds obscure errors) (IN::GDS_priority_t)

C:

```
GDS_status_t GDS_register_local_error_check(GDS_gds_t gds,
  GDS_check_func_t check_func,
  GDS_priority_t check_priority);
```

Fortran:

```
GDS_REGISTER_LOCAL_ERROR_CHECK(gds, check_func, check_priority, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  TYPE(C_FUNPTR), VALUE, INTENT(IN) :: check_func
  INTEGER, VALUE, INTENT(IN) :: check_priority
  INTEGER, INTENT(OUT) :: status
```

GDS_register_global_error_check (*gds, error_check_function, check_priority*)

User programs can implement whatever error checking they would like, and then signal errors to the GVR system using raise_error. The *register_global_error_check()* function is a collective call, which allows users to register checking function to all the processes for the given GDS. It is assumed these checking routines are coordinated with all the processes. It will be triggered and run by the GVR system for each processes with synchronization. It is assumed these checking routines make appropriate calls to *raise_error()* when they find them. During an error-check, the application must not write to the passed GDS.

Parameters:

- **gds**: GDS on which this error checking routine is being registered (IN::GDS_gds_t)
- **error_check_function**: the checking function, which takes a GDS, the priority of the check, and a flag specifying whether or not the check is coordinated. The function is expected to check the given GDS for faults. If a fault is located, this function should call GDS_raise_error with the appropriate arguments. If this error check is coordinated, every process in the communicator for the given GDS must pass the same value for this argument (IN::GDS_status_t (*check_func) (GDS_gds_t gds, GDS_priority_t check_priority))
- **check_priority**: desired priority for the error check. High priority means run first (low cost, finds common errors). Medium and low priorities mean run as last resort (may be high cost, finds obscure errors) (IN::GDS_priority_t)

C:

```
GDS_status_t GDS_register_global_error_check(GDS_gds_t gds,
  GDS_check_func_t check_func,
  GDS_priority_t check_priority);
```

Fortran:

```
GDS_REGISTER_GLOBAL_ERROR_CHECK(gds, check_func, check_priority, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  TYPE(C_FUNPTR), VALUE, INTENT(IN) :: check_func
  INTEGER, VALUE, INTENT(IN) :: check_priority
  INTEGER, INTENT(OUT) :: status
```

GDS_raise_local_error (*gds, error_desc*)

Indicates to the runtime that a fault has occurred in the calling process. It will interrupt the calling process and t

- **gds**: the GDS object (IN::GDS_gds_t)
- **error_desc**: description of the error raised. In general, system-raised error types will be enumerated and available to all programs being run at initialization time.

Application programs must declare and then handle any errors that they raise (IN::GDS_error_t)

C:

```
GDS_status_t GDS_raise_local_error(GDS_gds_t gds, GDS_error_t error_desc);
```

Fortran:

```
GDS_RAISE_LOCAL_ERROR(gds, err_desc, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  TYPE(C_PTR), VALUE, INTENT(IN) :: err_desc
  INTEGER, INTENT(OUT) :: status
```

GDS_raise_global_error (*gds, error_desc*)

Indicates to the runtime that a fault has occurred that involves all the processes for the given GDS object. It will interrupt all the processes and trigger the global error handler. It is not a collective call, either local check or global check can raise global error. The processes using the given GDS object are prevented from performing GDS library operations on the GDS object until a *GDS_resume_global* is called.

- **gds**: the GDS object (IN::GDS_gds_t)
- **error_desc**: description of the error raised. In general, system-raised error types will be enumerated and available to all programs being run at initialization time. Application programs must declare and then handle any errors that they raise (IN::GDS_error_t)

Notes This function raises an global error, but the corresponding error handler will not be invoked immediately. The handler will be invoked at the next stable point (*i.e. GDS_fence* or *GDS_version_inc*).

C:

```
GDS_status_t GDS_raise_global_error(GDS_gds_t gds, GDS_error_t error_desc);
```

Fortran:

```
GDS_RAISE_GLOBAL_ERROR(gds, err_desc, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  TYPE(C_PTR), VALUE, INTENT(IN) :: err_desc
  INTEGER, INTENT(OUT) :: status
```

GDS_register_local_error_handler (*gds, pred, recovery_func*)

Specifies a local recovery procedure in the calling process. If a local error is detected, the runtime will execute the given function on the involved process.

Parameters:

- **gds**: a handle of the gds object for which the error handler will receive notifications (IN::GDS_gds_t)
- **pred**: indicates an error matching predicates to describe what kind of errors *recovery_func* will handle (IN::GDS_error_pred_t)
- **recovery_func**: the specified recovery function. The function is expected to take a GDS object, a boolean specifying whether or not this error recovery is coordinated, and an object describing the error, and return a status signifying whether or not the recovery has succeeded. Component-specific error information, such as location information for a memory error, shall be encapsulated in error_desc. The recover_func may assume that the application has been halted, and is expected to call GDS_resume on

its given GDS after successful recovery (IN::GDS_status_t (*recover_func)(GDS_gds_t gds, boolean is_coordinated, GDS_error_t error_desc))

Notes It is safe to free the *pred* object by calling *GDS_free_error_pred* when this function returns.

C:

```
GDS_status_t GDS_register_local_error_handler(GDS_gds_t gds,
  GDS_error_pred_t pred, GDS_recovery_func_t recovery_func);
```

Fortran:

```
GDS_REGISTER_LOCAL_ERROR_HANDLER(gds, pred, recovery_func, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, VALUE, INTENT(IN) :: pred
  TYPE(C_FUNPTR), INTENT(IN) :: recovery_func
  INTEGER, INTENT(OUT) :: status
```

GDS_register_global_error_handler (*gds, pred, recovery_func*)

Specifies a globally coordinated recovery procedure in all the processes for a given GDS. It is a collective call. If an error is detected within the given component, then the runtime will execute the given function in coordinated manner at the future stable point.

Parameters:

- **gds**: a handle of the gds object for which the error handler will receive notifications (IN::GDS_gds_t)
- **pred**: indicates an error matching predicates to describe what kind of errors *recovery_func* will handle (IN::GDS_error_pred_t)
- **recovery_func**: the specified recovery function. The function is expected to take a GDS object, a boolean specifying whether or not this error recovery is coordinated, and an object describing the error, and return a status signifying whether or not the recovery has succeeded. Component-specific error information, such as location information for a memory error, shall be encapsulated in error_desc. The recover_func may assume that the application has been halted, and is expected to call GDS_resume on its given GDS after successful recovery (IN::GDS_status_t (*recover_func)(GDS_gds_t gds, boolean is_coordinated, GDS_error_t error_desc))

Notes It is safe to free the *pred* object by calling *GDS_free_error_pred* when this function returns.

C:

```
GDS_status_t GDS_register_global_error_handler(GDS_gds_t gds,
  GDS_error_pred_t pred, GDS_recovery_func_t recovery_func);
```

Fortran:

```
GDS_REGISTER_GLOBAL_ERROR_HANDLER(gds, pred, recovery_func, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  TYPE(C_PTR), VALUE, INTENT(IN) :: pred
  TYPE(C_FUNPTR), VALUE, INTENT(IN) :: recovery_func
  INTEGER, INTENT(OUT) :: status
```

GDS_resume_local (*gds, error_desc*)

Resumes the process if the local error handler succeeds.

Parameters:

- **gds**: the GDS this error handler is resuming (IN::GDS_gds_t)

- **error_desc**: the error descriptor passed to the error handler function (IN::GDS_error_t)

C:

```
GDS_status_t GDS_resume_local(GDS_gds_t gds, GDS_error_t error_desc);
```

Fortran:

```
GDS_RESUME_LOCAL(gds, err_desc, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  TYPE(C_PTR), VALUE, INTENT(IN) :: err_desc
  INTEGER, INTENT(OUT) :: status
```

GDS_resume_global (*gds, error_desc*)

Resumes the processes if the global error handler succeeds.

Parameters:

- **gds**: the GDS this error handler is resuming (IN::GDS_gds_t)
- **error_desc**: the error descriptor passed to the error handler function (IN::GDS_error_t)

C:

```
GDS_status_t GDS_resume_global(GDS_gds_t gds, GDS_error_t error_desc);
```

Fortran:

```
GDS_RESUME_GLOBAL(gds, err_desc, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  TYPE(C_PTR), VALUE, INTENT(IN) :: err_desc
  INTEGER, INTENT(OUT) :: status
```

GDS_check_all_errors (*gds*)

Trigger all of the registered global/local error check functions and gather the checking results from all processes. It is a collective function.

Parameters:

- **gds**: the GDS on which error checking routine is registered (IN::GDS_gds_t)

C:

```
GDS_status_t GDS_check_all_errors(GDS_gds_t gds);
```

Fortran:

```
GDS_CHECK_ALL_ERRORS(gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_invoke_local_error_handler (*gds*)

Invoke the registered local error handler.

Parameters:

- **gds**: the GDS on which error handler is registered (IN::GDS_gds_t)

C:

```
GDS_status_t GDS_invoke_local_error_handler(GDS_gds_t gds);
```

Fortran:

```
GDS_INVOKE_LOCAL_ERROR_HANDLER(gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_invoke_global_error_handler (*gds*)

Invoke the registered global error handler from all processes. By calling this function, application defines a stable point. It is a collective function.

Parameters:

- **gds**: the GDS on which error handler is registered (IN::GDS_gds_t)

C:

```
GDS_status_t GDS_invoke_global_error_handler(GDS_gds_t gds);
```

Fortran:

```
GDS_INVOKE_GLOBAL_ERROR_HANDLER(gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

6.4.3 Version Navigation

GDS_get_version_number (*gds, version_number*)

Returns version number for the given GDS object.

Parameters:

- **gds**: the GDS object whose version that should be queried (IN::GDS_gds_t)
- **version_number**: the version number of the given GDS object (OUT::GDS_size_t)

C:

```
GDS_status_t GDS_get_version_number(GDS_gds_t gds, GDS_size_t *version_number);
```

Fortran:

```
GDS_GET_VERSION_NUMBER(gds, ver_num, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER(8), INTENT(OUT) :: ver_num
  INTEGER, INTENT(OUT) :: status
```

GDS_get_version_label (*gds, label, label_size*)

Returns version label and its size for the given GDS object.

Parameters:

- **gds**: the GDS object whose version that should be queried (IN::GDS_gds_t)
- **label**: the version label of the given GDS object (OUT::string)
- **label_size**: the size of version label (OUT::size_t)

C:

```
GDS_status_t GDS_get_version_label(GDS_gds_t gds, char **label, size_t *label_size);
```

Fortran:

```
GDS_GET_VERSION_LABEL(gds, label, label_size, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  CHARACTER, POINTER, INTENT(OUT) :: label(:)
  INTEGER(8), INTENT(OUT) :: label_size
  INTEGER, INTENT(OUT) :: status
```

GDS_move_to_newest (*gds*)

Sets the version of the GDS object to its most recent available version.

Parameters:

- **gds**: the GDS object (INOUT::GDS_gds_t)

C:

```
GDS_status_t GDS_move_to_newest(GDS_gds_t gds);
```

Fortran:

```
GDS_MOVE_TO_NEWEST(gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_move_to_next (*gds*)

Sets the version of the given GDS object to the version of that GDS that is both further forward in time than the given version, and is the oldest available version that is forward in time.

Parameters:

- **gds**: the GDS object (INOUT::GDS_gds_t)

Returns: an error if the given version of the GDS is already the newest version.

C:

```
GDS_status_t GDS_move_to_next(GDS_gds_t gds);
```

Fortran:

```
GDS_MOVE_TO_NEXT(gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_move_to_prev (*gds*)

Sets the version of the given GDS object to the version of that GDS that is both backward in time from the given version, and the newest available version that is backward in time.

Parameters:

- **gds**: the GDS object (INOUT::GDS_gds_t)

Returns an error if the given version of the GDS is already the oldest version.

C:

```
GDS_status_t GDS_move_to_prev(GDS_gds_t gds);
```

Fortran:

```
GDS_MOVE_TO_PREV(gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER, INTENT(OUT) :: status
```

GDS_version_dec (*gds, dec*)

Sets the GDS_handle to point to current minus dec. If the requested version doesn't exist, returns an error.

Parameters:

- **gds**: the GDS handle where the version will be decremented (IN::GDS_gds_t)
- **dec**: number by which to decrement the version number (IN::nonnegative integer)

C:

```
GDS_status_t GDS_version_dec(GDS_gds_t gds, GDS_size_t dec);
```

Fortran:

```
GDS_VERSION_DEC(gds, dec, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: gds
  INTEGER(8), VALUE, :: dec
  INTEGER, INTENT(OUT) :: status
```

GDS_descriptor_clone (*in_gds, clone_gds*)

Returns a copy of the given GDS descriptor. The returned descriptor should be freed by GDS_free when no longer needed.

Parameters:

- **in_gds**: the GDS descriptor to be cloned (IN::GDS_gds_t)
- **clone_gds**: GDS descriptor identical to in_gds (OUT::GDS_gds_t)

C:

```
GDS_status_t GDS_descriptor_clone(GDS_gds_t in_gds, GDS_gds_t *clone_gds);
```

Fortran:

```
GDS_DESCRIPTOR_CLONE(in_gds, clone_gds, status)
  TYPE(C_PTR), VALUE, INTENT(IN) :: in_gds
  TYPE(C_PTR), INTENT(OUT) :: clone_gds
  INTEGER, INTENT(OUT) :: status
```

6.5 GDS Types

Predefined Constants for Type: GDS_comm_t

- **GDS_COMM_WORLD**: communicator containing all processes using the GDS library.

Predefined Constants for Type: GDS_priority_t

- **GDS_PRIORITY_HIGH** or **GDS_PRIORITY_CRITICAL**: these structures are critical for recovery, and may not be reconstructible. If lost, the computation will need to be restarted.
- **GDS_PRIORITY_MEDIUM** or **GDS_PRIORITY_EXPENSIVE**: these structures are costly to diagnose and reconstruct. It is worth expending redundancy effort to protect these structures.

- **GDS_PRIORITY_LOW** or **GDS_PRIORITY_CHEAP**: these structures can be restored inexpensively. Perhaps they are replicated across nodes, constants loaded in to the program, or can be reconstructed easily via symmetry or other semantics or with inexpensive computation.

Because error handlers can be defined by applications and registered with the GVR runtime, it is straightforward to implement varied error reporting/filtering. Lower level error handling routines can simply filter the requests. For example, one could register promiscuous filters for critical structures, and more opaque filters for lower priority structures.

Predefined Constants for Type: GDS_gds_t

- **GDS_ROOT**

Special GDS object which can be passed only to error-checking, error-handling, and synchronization functions. Specifies that the operation in question should be applied to all GDS objects currently registered with the library.

6.5.1 GDS types corresponding to MPI types

Elements of the following GDS datatypes can be safely cast to elements of their corresponding MPI datatypes and vis

GDS type	MPI type
GDS_comm_t	MPI_Comm
GDS_datatype_t	MPI_Datatype
GDS_op_t	MPI_Op
GDS_info_t	MPI_Info

6.6 Scraps

Placeholder: Distribution-Related Operations

These operations will manipulate objects of type GDS_distrib_t, which express global data layout (including global gds size?)

6.7 Revision History

6.7.1 API 1.0.0

- Introduces new Open Resilience error handling APIs
  - "Error category" was deprecated
  - A notion of "error matching predicate" was introduced, and *GDS_register_*_error_handler* now takes an error matching predicate object.
  - *GDS_create_error_descriptor* was changed. Attributes are now added via *GDS_add_error_attr* function.
- API changes
  - *GDS_create* now takes *GDS_info_t info* instead of *const char *hint*
  - *GDS_BASE* was removed from *GDS_attr_t*
  - Supported datatypes are clarified for *GDS_compare_and_swap*
  - *GDS_check_all_error* was renamed to *GDS_check_all_errors*

  - A new API *GDS_get_version_label* was introduced
  - *GDS_enumerate_all_versions* was removed

6.7.2 API 0.7.5

- Adds Fortran APIs documentation

6.7.3 API 0.7.4

- Introduces pre-defined error categories
- Introduces `GDS_create_error_descriptor` and `GDS_extend_error_category` functions
- `GDS_resume_*` now takes an error descriptor as an argument
- `GDS_get_error_attr` now takes `GDS_error_attr_t` as an attribute key type instead of `GDS_attr_t`.

6.7.4 API 0.7.3

- Introduces GDS_info_t.
- Type of the info argument for GDS_create and GDS_alloc has been changed from string to GDS_info_t.
- GDS_create and GDS_alloc now supports both row-major and column-major order.