CONTENTS

1 Release Notes ........................................... 1
  1.1 What’s new in this version .......................... 1
  1.2 Prerequisites ........................................ 2
  1.3 How to build & install ............................... 3
  1.4 Known issues and restrictions ...................... 3

2 Introduction ............................................. 5
  2.1 Concepts ............................................. 5
  2.2 Objectives ........................................... 5
  2.3 Design Requirements ................................. 6

3 Getting Started .......................................... 7
  3.1 Initialization ......................................... 7
  3.2 Initialization with MPI library ..................... 7
  3.3 How to build your program ......................... 8
  3.4 How to run your program ........................... 8
  3.5 Code example ......................................... 8

4 Use Cases .................................................. 11
  4.1 Introduction ......................................... 11
  4.2 Parallel program structure ......................... 11
  4.3 Case Studies: general computation and multi-version .... 13

5 The GVR Interface 0.7.5 ................................. 21
  5.1 Initialization ......................................... 21
  5.2 Creating a Global Data Structure ................. 22
  5.3 Using a Global Data Structure .................... 26
  5.4 Versioning, Error-Signaling, Error-Checking, and Error-Recovery .... 31
  5.5 GDS Types ............................................ 41
  5.6 Scraps ................................................. 42
  5.7 Revision History ..................................... 42
CHAPTER ONE

RELEASE NOTES

1.1 What’s new in this version

1.1.1 0.8.1-rc0

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

1.1.2 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.

1.1.3 0.7.1

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

1.1.4 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.
1.1.5 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

1.1.6 0.5.3

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

1.1.7 0.5.2

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

1.1.8 0.5.1

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

1.1.9 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

1.2 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 Ubuntu 12.04 amd64, gcc 4.6.3, and mpich 3.0.4.
1.3 How to build & install

GVR requires LRDS (Local Reliable Data Storage) as an external library. First, build and install LRDS.

Then, the installation process of GVR is quite straightforward; type three commands.

$ ./configure
$ 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.

1.4 Known issues and restrictions

Currently the following features are not supported.

- `GDS_create` is not implemented
- `GDS_get_comm` is not implemented
- `GDS_get_acc` is not implemented
- `GDS_compare_and_swap` is not implemented
- `GDS_version_inc` currently ignores `label` and `label_size` arguments.
- Direct buffer access functions, `GDS_access`, `GDS_get_access_buffer_type`, `GDS_release` are not implemented
- Global error recovery functions are not invoked when `GDS_fence` is called with `NULL`.
- **Support for error checking is limited.**
  - error checking for external events is not supported

Currently we have the following restrictions.

- All the versions are kept in memory.
- Currently `GDS_THREAD_SINGLE` is the only supported thread execution model.
- Due to a bug, Fortran testing code (tests/tests_f90) crashes when being executed with more than 1 process.
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.
GETTING STARTED

3.1 Initialization

The simplest GVR program looks like this:

```c
#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.

```c
#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. Make sure to specify -L and -lgds options.

$ mpicc ... -I<path/to/gvr-installdir>/include -L<path/to/gvr-installdir>/lib <your-gvr-program>.c

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

$ mpif90 ... -I<path/to/gvr-installdir>/include -L<path/to/gvr-installdir>/lib <your-gvr-program>

3.3.1 Note for static linking

If you want to link libgds.a statically to your program, you’ll 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<path/to/gvr-installdir>/include -L<path/to/gvr-installdir>/lib -static <your-gvr-program>

3.4 How to run your program

Current GVR implementation is built on top of MPI library, so you can run your GVR-applied 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=<path/to/gvr-installdir>/lib mpiexec -n 4 <your-gvr-program>

3.5 Code example

Writing Fortran programs requires users having basic knowledge of interoperability with C, especially handling with C pointer and function pointer. For more details, please refer to here. 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)
call GDS_PUT(C_LOC(put_buf), ld, lo_idx, hi_idx, g, stat) ! Use C_LOC to get buffer address
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
4.1 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.

4.2 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.
4.2.1 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)

4.2.2 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)

4.2.3 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

4.2.4 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?

4.2.5 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?

4.3 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".

4.3.1 Case A1: 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).

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

4.3.2 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
end for
GDS_put(my_c, [me, [0, N-1]], ..., gds_C)
GDS_fence(gds_C)
/* free gds_A, gds_B, and gds_C */
...

4.3.3 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 \(\text{partition}(i)\), 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 \(\text{cols}(\text{rows})\) return the set of columns for which at least one entry is nonzero given a set of rows. Let \(\text{norm1}(\text{gds})\) 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) * column_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_fence(gds_x)
  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

4.3.4 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.
 */

4.3. Case Studies: general computation and multi-version
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
    info_buf := calculate_node_info(current_node, info_buf)
    GDS_put(info_buf, info_pertaining_to(current_node), ..., gds_info)
    GDS_fence(gds_info)
  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
    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).
*/

if repartition_is_required() then
    GDS_put(position_buf, my_particles, ..., gds_x)
    GDS_fence(gds_tree)
    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

4.3.5 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 occurred 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

4.3.6 Case studies: error check, signaling, and recovery

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

miniFE

function MAIN
  /* create A and b */
  ...
  /* initialize x */
  ...
  GDS_alloc(..., gds_x)
  /* Register local error handler return iter solve(A, b, x, gds_x) */
  GDS_register_local_error_handler(gds_x, ..., reload)
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_put_error_attr(err_desc, LOCAL_BUFFER_KEY, x)
        /* Trigger the local error handler */
        GDS_raise_local_error(gds_x, error_desc)
        do_necessary_recalculation()
    end if
    if iteration % CP_INTERVAL == 0 then
        GDS_put(x, ..., gds_x)
        GDS_version_inc(gds_x, ...)
    end if
    x := do_calculation(A, b, x)
    until normr < TOL_2
end function

function MAIN
    GDS_alloc(GDS_PRIORITY_HIGH, gds)
    GDS_register_global_error_handler(gds, rollback)
    /* Register global error handler */
    GDS_register_global_error_check(gds, full_check)
    return iter_computation() /* Register global error check for rollback */
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()
end function

function iter_computation
    repeat
        do_comp_and_comm(atoms)
        if atoms_out_of_box(atoms) then
            /* Raise global error triggers the error handler. */
            GDS_raise_global_error(gds, global_error_desc)
        end if
        GDS_put(atoms, gds)
        GDS_version_inc(gds)
        until converge()
    end function

4.3. Case Studies: general computation and multi-version
5.1 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(IN) :: size
  INTEGER, INTENT(OUT) :: status

5.2 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 scaler 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 documentation 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)

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`.

5.2. Creating a Global Data Structure
**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. Application uses GDS as a remote access array only.

**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:**

```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:**

```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:**

```c
GDS_status_t GDS_free(GDS_gds_t gds)
```
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. The following attributes are predefined and are required to be present:

- GDS_TYPE
- GDS_CREATE_FLAVOR
- GDS_BASE
- GDS_GLOBAL_LAYOUT

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_CREATE_FLAVOR: indicates how the GDS object was created. One of: GDS_FLAVOR_CREATE, GDS_FLAVORALLOC
- GDS_BASE: the local buffer address for the GDS object, if there is a local piece. Otherwise NULL.
- GDS_GLOBAL_LAYOUT: the global layout of the GDS.
- GDS_NUMBER_DIMENSIONS: a GDS_size_t signifying the number of dimensions of the GDS.
- 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.
- 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.
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:

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

### 5.3 Using a Global Data Structure

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

Returns an error if applied to a version of the gds that is not the current version.

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.

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:

```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:

```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 nonblock-
ing. wait() 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:

```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:

```fortran
GDS_GET(orig_addr, orig_ld, lo_idx, hi_idx, gds, 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.

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)
- **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.

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::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);

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

Compare and swap. 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.
• **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:

```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);
```

**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)

C:

```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)
```

**GDS_get_access_buffer_type (access_handle, buffer_type)**

### 5.3. Using a Global Data Structure

29
Returns a value signifying the nature of the buffer returned by \textit{GDS_access}.

**Parameters:**

- \textbf{access\_handle}: handle for the access operation that should be queried (IN::GDS_access\_handle\_t)
- \textbf{buffer\_type}: the nature of the buffer returned in the call to access. Should be one of: \textit{GDS\_ACCESS\_BUFFER\_DIRECT} or \textit{GDS\_ACCESS\_BUFFER\_COPY} (OUT::GDS_access\_buffer\_t)

C:
\begin{verbatim}
GDS_status_t GDS_get_access_buffer_type(GDS_access_handle_t access_handle, GDS_access_buffer_t buffer_type);
\end{verbatim}

**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:**

- \textbf{access\_handle}: handle that was output by a particular call to GDS_access (IN-OUT::GDS_access\_handle\_t)

C:
\begin{verbatim}
GDS_status_t GDS_release(GDS_access_handle_t access_handle);
\end{verbatim}

**GDS\_fence (gds)**

A collective operation. All read operations following the fence will reflect all of the write operations preceding it. If the programmer wishes to strongly separate read operations from writes after the fence, he or she may call wait in conjunction with fence.

**Parameters:**

- \textbf{gds}: the GDS object on which the fence takes place (IN::GDS_gds\_t)

C:
\begin{verbatim}
GDS_status_t GDS_fence(GDS_gds_t *gds);
\end{verbatim}

Fortran:
\begin{verbatim}
GDS_FENCE(gds, status)
   TYPE(C_PTR), VALUE, INTENT(IN) :: gds
   INTEGER, INTENT(OUT) :: status
\end{verbatim}

**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:**

- \textbf{gds}: the GDS object on which the wait takes place (IN::GDS_gds\_t)

C:
\begin{verbatim}
GDS_status_t GDS_wait(GDS_gds_t *gds);
\end{verbatim}
5.4 Versioning, Error-Signaling, Error-Checking, and Error-Recovery

5.4.1 Demarcating Versions

GDS_version_inc (gds, increment, label)

Advances the version for the given GDS object. Semantically equivalent to all clients calling gds_wait() followed by gds_fence() whether or not the runtime chooses to actually create a new version of the GDS object.

Parameters:

• gds: the GDS object (INOUT::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)

C:

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

Fortran:

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

Errors in GVR are described based on error categories, such as memory error category or CPU error category. Each error category is associated with several error attributes. An error attribute is a key-value pair that gives a parameter to describe the details of the error. For example, GDS_ERROR_MEMORY category is associated with two attributes, GDS_ERROR_MEMORY_OFFSET and GDS_ERROR_MEMORY_COUNT to describe the damaged memory region.

When an error occurs, a GVR program or GVR runtime library creates an error descriptor based on a category. A descriptor has error attributes defined in the category. A C++/Java programmer would imagine that an error category is a class and an error descriptor is an instance (object) of that class. Error descriptors are immutable objects, meaning that once the descriptor is created, its contents will never modified.

Error categories form a tree hierarchy. This means that when an error category B is a descendant of a category A, an error descriptor describing the category B can be also interpreted as a descriptor of the category A. A descendant category has all the attributes defined in the ascendants. The following error categories are pre-defined.

- GDS_ERROR_ROOT
  - GDS_ERROR_MEMORY
  - GDS_ERROR_NIC
  - GDS_ERROR_CPU
  - GDS_ERROR_NODE

GDS_ERROR_ROOT
The root error category. An ascendant of any other error categories.

Attributes:
- GDS_ERROR_CATEGORY: describes an error category to which this error descriptor belongs (GDS_error_category_t)

GDS_ERROR_MEMORY
Describes a memory data error.

Attributes:
- GDS_ERROR_MEMORY_OFFSET: starting index of the damaged region, represented in a 1-d coordinates (GDS_size_t)
- GDS_ERROR_MEMORY_COUNT: size of the damages region, represented in a 1-d coordinates (GDS_size_t)
API Functions

**GDS_extend_error_category** (parent, attr_keys, attr_lens, nAttrs, new_category)

Registers a new error category, deriving from one of the existing error category. This function is a collective call across the entire process in the program.

**Parameters:**
- **parent**: a parent error category (IN::GDS_error_category_t)
- **attr_keys**: an array of nAttrs elements of error attribute keys to add in this category. The runtime system will assign new attribute keys and store them in this array. (OUT::GDS_error_attr_t[])
- **attr_lens**: an array of nAttrs elements, describing maximum length of values associated with newly defined attributes. (IN::GDS_size_t[])
- **nAttrs**: number of attributes to define. Denotes the length of attr_keys and attr_lens. (IN::GDS_size_t)
- **new_category**: a newly defined category is returned (OUT::GDS_error_category_t)

**C:**

```c
GDS_status_t GDS_extend_error_category(
    GDS_error_category_t parent,
    GDS_error_attr_t *attr_keys, const GDS_size_t *attr_lens, GDS_size_t nAttrs,
    GDS_error_category_t *new_category);
```

**Fortran:**

```fortran
GDS_EXTEND_ERROR_CATEGORY(parent, attr_keys, attr_lens, nAttrs, new_category, status)
```

**GDS_create_error_descriptor** (category, attr_keys, attr_vals, nAttrs, error_desc)

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

**Parameters:**
- **category**: specifies an error category to which the new error descriptor belongs (IN::GDS_error_category_t)
- **attr_keys**: an array containing attribute keys to associate with this error (IN::GDS_error_attr_t[])
- **attr_vals**: an array containing pointers to attribute values to associate with this error. Values should not contain pointer, unless you are sure that the descriptor won’t be passed to another process. (IN::void **)
- **nAttrs**: number of attribute key-value pairs (IN::GDS_size_t)
- **error_desc**: returned handle to created error descriptor (OUT::GDS_error_t)

**C:**

```c
```
GDS_create_error_descriptor(GDS_error_category_t category,
   const GDS_error_attr_t *attr_keys, void * const attr_vals[], GDS_size_t n_attrs,
   GDS_error_t *error_desc)

Fortran:
GDS_CREATE_ERROR_DESCRIPTOR(category, attr_keys, attr_vals, nattrs, error_desc, status)
   INTEGER, VALUE, INTENT(IN) :: category
   INTEGER, DIMENSION(*) , INTENT(IN) :: attr_keys
   TYPE(C_PTR), VALUE, INTENT(IN) :: attr_vals
   INTEGER(*), VALUE, INTENT(IN) :: nattrs
   TYPE(C_PTR), INTENT(OUT) :: error_desc
   INTEGER, INTENT(OUT) :: status

GDS_get_error_attr error_desc, attribute_key, attribute_value, flag)
Queries attributes of the given error description object. If the specified attribute is present, the
flag argument is set to true. The following attributes are predefined and are required to be present:

• GDS_ERROR_CATEGORY

Parameters:

• error_desc: the error description (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_gds_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_register_local_error_check (gds, error_category, 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_category: the category of error to check (IN::GDS_error_category_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:

```c
GDS_status_t GDS_register_local_error_handler(GDS_gds_t gds,
    GDS_error_t error_desc, GDS_recovery_func_t recovery_func);
```

Fortran:

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

**GDS_register_global_error_check (gds, error_category, 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_category**: the category of error to check (IN::GDS_error_category_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, GDSPriority_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:

```c
GDS_status_t GDS_register_global_error_handler(GDS_gds_t gds,
    GDS_error_t error_desc, GDS_recovery_func_t recovery_func,
    GDS_priority_t check_priority);
```

**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 an

- **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:

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

Fortran:

```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)

C:

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

**GDS_register_local_error_handler (gds, error_category, 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)

• **error_category**: this argument indicates one of a set of error categories (IN::GDS_category_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))

C:

```c
GDS_status_t GDS_register_local_error_handler(GDS_gds_t gds,
                                           GDS_category_t error_category, GDS_recovery_func_t recovery_func);
```

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

GDS_register_global_error_handler (gds, error_category, 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)
- **error_category**: this argument indicates one of a set of error types (IN::GDS_error_category_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))

C:

```c
GDS_status_t GDS_register_global_error_handler(GDS_gds_t gds,
                                             GDS_error_category_t error_category,
                                             GDS_recovery_func_t recovery_func);
```

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:

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

Fortran:

```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_error (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_error(GDS_gds_t gds);

**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_global_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
5.4.3 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:

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

Fortran:

```fortran
GDS_GET_VERSION_NUMBER(gds, ver_num, status)
```

Type(C_PTR), Value, Intent(IN) :: gds
Inte(8), Intent(OUT) :: ver_num
Inte, 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:

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

Fortran:

```fortran
GDS_MOVE_TO_NEWEST(gds, status)
```

Type(C_PTR), Value, Intent(IN) :: gds
Inte, 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:

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

Fortran:

```fortran
GDS_MOVE_TO_NEXT(gds, status)
```

Type(C_PTR), Value, Intent(IN) :: gds
Inte, 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:

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

Fortran:

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

**GDS Enumerate All Versions** (gds, gds_version_list)

Returns a list of existing versions for the specified GDS objects.

**Parameters:**

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

  - **gds_version_list**: list of GDS descriptors (OUT::list of GDS_gds_t)

C:

```c
GDS_status_t GDS_enumerate_all_versions(GDS_gds_t gds, GDS_gds_t *gds_version_list);
```

**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 (INOUT::GDS_gds_t)

  - **dec**: number by which to decrement the version number (IN::nonnegative integer)

C:

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

Fortran:

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

```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

5.5 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_error_category_t

- GDS_ERROR_MEMORY
- GDS_ERROR_NIC
- GDS_ERROR_CPU
- GDS_ERROR_NODE

Range of error categories for which error handlers can be registered. We expect this is a growing list of error types which can be augmented and tracked by the runtime, and appropriate error handlers can be added.

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.
5.5.1 GDS types corresponding to MPI types

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

<table>
<thead>
<tr>
<th>GDS type</th>
<th>MPI type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDS_comm_t</td>
<td>MPI_Comm</td>
</tr>
<tr>
<td>GDS_datatype_t</td>
<td>MPI_Datatype</td>
</tr>
<tr>
<td>GDS_op_t</td>
<td>MPI_Op</td>
</tr>
<tr>
<td>GDS_info_t</td>
<td>MPI_Info</td>
</tr>
</tbody>
</table>

5.6 Scraps

Placeholder: Distribution-Related Operations

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

5.7 Revision History

5.7.1 API 0.7.5

- Adds Fortran APIs documentation

5.7.2 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`

5.7.3 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.