YGM

YGM Developers

Jun 30, 2025

CONTENTS:

CHAPTER

ONE

GETTING STARTED

1.1 What is YGM?

YGM is an asynchronous communication library written in C++ and designed for high-performance computing (HPC) use cases featuring irregular communication patterns. YGM includes a collection of distributed-memory storage containers designed to express common algorithmic and data-munging tasks. These containers automatically partition data, allowing insertions and, with most containers, processing of individual elements to be initiated from any runninng YGM process.

Underlying YGM's containers is a communicator abstraction. This communicator asynchronously sends messages spawned by senders with receivers needing no knowledge of incoming messages prior to their arrival. YGM communications take the form of *active messages*; each message contains a function object to execute (often in the form of C++ lambdas), data and/or pointers to data for this function to execute on, and a destination process for the message to be executed at.

YGM also includes a set of I/O primitives for parsing collections of input documents in parallel as independent lines of text and streaming output lines to large numbers of destination files. Current parsing functionality supports reading input as CSV, ndjson, and unstructured lines of data.

1.2 General YGM Operations

YGM is built on its ability to communicate active messages asynchronously between running processes. This does not capture every operation that can be useful, for instance collective operations are still widely needed. YGM uses prefixes on function names to distinguish their behaviors in terms of the processes involved. These prefixes are:

- `async_`: Asynchronous operation initiated on a single process. The execution of the underlying function may occur on a remote process.
- `local_`: Function performs only local operations on data of the current process. In uses within YGM containers with partitioning schemes that determine item ownership, care must be taken to ensure the process a `local_` operation is called from aligns with the item's owner. For instance, calling `ygm::container::map::local_insert` will store an item on the process where the call is made, but the `ygm::container::map` may not be able to look up this location if it is on the wrong process.
- No Prefix: Collective operation that must be called from all processes.

The primary workhorse functions in YGM fall into the two categories of `async_` and `for_all` operations. In an `async_` operation, a lambda is asynchronously sent to a (potentially) remote process for execution. In many cases with YGM containers, the lambda being executed is not provided by the user and is instead part of the function itself, e.g. `async_insert` calls on most containers. A `for_all` operation is a collective operation in which a lambda is executed locally on every process while iterating over all locally held items of some YGM object. The items iterated over can be items in a YGM container, items coming from a map, filter, or flatten applied to a container, or all lines in a collection of files in a YGM I/O parser.

1.2.1 Lambda Capture Rules

Certain `async_` and `for_all` operations require users to provide lambdas as part of their executions. The lambdas that can be accepted by these two classes of functions follow different rules pertaining to the capturing of variables:

- `async_` calls cannot capture (most) variables in lambdas. Variables necessary for lambda execution must be provided as arguments to the `async_` call. In the event that the data for the lambda resides on the remote process the lambda will execute on, a `ygm::ygm_ptr` should be passed as an argument to the `async_`.
- `for_all` calls assume lambdas take only the arguments inherently provided by the YGM object being iterated over. All other necessary variables *must* be captured. The types of arguments provided to the lambda can be identified by the `for_all_args` type within the YGM object.

These differences in behavior arise from the distinction that `async_` lambdas may execute on a remote process, while `for_all` lambdas are guaranteed to execute locally to a process. In the case of `async_` operations, the lambda and all arguments must be serialized for communication, but C++ does not provide a method for inspection of variables captured in the closure of a lambda. In the case of `for_all` operations, the execution is equivalent to calling std::for_each on entire collection of items held locally.

1.3 Requirements

- C++20 - GCC versions 11 and 12 are tested. Your mileage may vary with other compilers.
- Cereal - C++ serialization library
- MPI
- Optionally, Boost 1.77 to enable Boost.JSON support.

1.4 Using YGM with CMake

YGM is a header-only library that is easy to incorporate into a project through CMake. Adding the following to CMakeLists.txt will install YGM and its dependencies as part of your project:

```
set(DESIRED_YGM_VERSION 0.6)
find_package(ygm ${DESIRED_YGM_VERSION} CONFIG)
if (NOT ygm_FOUND)
    FetchContent_Declare(
        ygm
        GIT_REPOSITORY https://github.com/LLNL/ygm
        GIT_TAG v${DESIRED_YGM_VERSION}
    )
    FetchContent_GetProperties(ygm)
    if (ygm_POPULATED)
        message(STATUS "Found already populated ygm dependency: "
                       ${ygm_SOURCE_DIR}
        )
    else ()
        set(JUST_INSTALL_YGM ON)
```

(continues on next page)

(continued from previous page)

```
        set(YGM_INSTALL ON)
        FetchContent_Populate(ygm)
        add_subdirectory(${ygm_SOURCE_DIR} ${ygm_BINARY_DIR})
        message(STATUS "Cloned ygm dependency " ${ygm_SOURCE_DIR})
    endif ()
else ()
    message(STATUS "Found installed ygm dependency " ${ygm_DIR})
endif ()
```

1.5 License

YGM is distributed under the MIT license.

All new contributions must be made under the MIT license.

See LICENSE-MIT, NOTICE, and COPYRIGHT for details.

SPDX-License-Identifier: MIT

1.6 Release

LLNL-CODE-789122

CHAPTER

TWO

YGM::COMM CLASS REFERENCE.

2.1 Communicator Overview

The communicator `ygm::comm` is the central object in YGM. The communicator controls an interface to an MPI communicator, and its functionality can be modified by additional optional parameters.

Communicator Features:

- **Message Buffering** - Increases application throughput at the expense of increased message latency.
- **Message Routing** - Extends benefits of message buffering to extremely large HPC allocations.
- **Fire-and-Forget RPC Semantics** - A sender provides the function and function arguments for execution on a specified destination rank through an *async* call. This function will complete on the destination rank at an unspecified time in the future, but YGM does not explicitly make the sender aware of this completion.

2.2 Communicator Hello World

Here we will walk through a basic "hello world" YGM program. The examples directory in the YGM tutorial contains several other examples, including many using YGM's storage containers.

To begin, headers for a YGM communicator are needed:

```
#include <ygm/comm.hpp>
```

At the beginning of the program, a YGM communicator must be constructed. It will be given `argc` and `argv` like `MPI_Init`.

```
ygm::comm world(&argc, &argv);
```

Next, we need a lambda to send through YGM. We'll do a simple hello_world type of lambda.

```
auto hello_world_lambda = [](const std::string &name) {
        std::cout << "Hello " << name << std::endl;
};
```

Finally, we use this lambda inside of our *async* calls. In this case, we will have rank 0 send a message to rank 1, telling it to greet the world

```
if (world.rank0()) {
        world.async(1, hello_world_lambda, std::string("world"));
}
```

A full, compilable version of this example is found here.

2.2.1 ygm::comm

class **comm**

Public Functions

inline **comm**(int *argc, char ***argv)

YGM communicator constructor.

```
#include <ygm/comm.hpp>

int main(int argc, char **argv) {
   ygm::comm world(&argc, &argv);
}
```

Parameters

- **argc** – Pointer to number of arguments given to command line
- **argv** – Pointer to array of command line arguments

Returns

Constructed *ygm::comm* object using MPI_COMM_WORLD for communication

inline **comm**(MPI_Comm comm)

YGM communicator constructor.

Parameters

mcomm – MPI communicator to use for underlying communication

Returns

Constructed *ygm::comm* object

inline **~comm**()

Destructor for comm object.

Calls a *barrier()* to ensure all messages have been processed, cancels all outstanding MPI receives and destroys MPI communicators set up for use within the *ygm::comm*

inline void **welcome**(std::ostream &os = std::cout)

Prints a welcome message with configuration details.

Prints a YGM welcome statement including information about internal YGM parameters.

Parameters

os – Output stream to print welcome message to

inline void **stats_reset**()

Resets counters within the comm_stats object being used by the *ygm::comm*.

Useful for separating information about communication performed in computation of interest from set-up or from other trials of the same experiment.

inline void `stats_print`(const std::string &name = "", std::ostream &os = std::cout)

Prints information about communication tracked in comm_stats object.

Parameters

- `name` – Label to be printed with stats
- `os` – Output stream to print stats to

template<typename `AsyncFunction`, typename ...`SendArgs`>
inline void `async`(int dest, *AsyncFunction* &&fn, const *SendArgs*&... args)

Asynchronous message initiation.

Serializes function object and queues for sending. Message will be sent and executed at some future time that YGM deems appropriate.

Template Parameters

- `AsyncFunction` – Type of function object
- `SendArgs...` – Variadic type of arguments to send along with function. All types must be serializable.

Parameters

- `dest` – Rank to execute function on
- `fn` – Function object to execute at remote destination
- `args...` – Variadic arguments to send with message and pass to function during execution

template<typename `AsyncFunction`, typename ...`SendArgs`>
inline void `async`(int dest, *AsyncFunction* &&fn, const *SendArgs*&... args) const

template<typename `AsyncFunction`, typename ...`SendArgs`>
inline void `async_bcast`(*AsyncFunction* &&fn, const *SendArgs*&... args)

Asynchronous message initiation for function that is sent to all ranks.

Serializes function object and queues for sending to all ranks. Message will be sent and executed at some future time that YGM deems appropriate. Messages are sent along an implicitly defined broadcast tree that takes advantage of knowledge of rank assignments to compute nodes.

Template Parameters

- `AsyncFunction` – Type of function object
- `SendArgs...` – Variadic type of arguments to send along with function. All types must be serializable.

Parameters

- `fn` – Function object to execute at remote destination
- `args...` – Variadic arguments to send with message and pass to function during execution

template<typename `AsyncFunction`, typename ...`SendArgs`>
inline void `async_bcast`(*AsyncFunction* &&fn, const *SendArgs*&... args) const

template<typename `AsyncFunction`, typename ...`SendArgs`>

inline void **async_mcast**(const std::vector<int> &dests, *AsyncFunction* &&fn, const *SendArgs*&... args)

template<typename **AsyncFunction**, typename ...**SendArgs**>
inline void **async_mcast**(const std::vector<int> &dests, *AsyncFunction* &&fn, const *SendArgs*&... args) const

inline void **cf_barrier**() const

Control Flow Barrier Only blocks the control flow until all processes in the communicator have called it. See: MPI_Barrier()

inline void **barrier**()

Full communicator barrier.

Collective operation that processes all messages (including any recursively produced messages) on all ranks. All ranks must complete their messages before any rank is able to return from the *barrier()* call.

inline void **barrier**() const

Full communicator barrier that can be called on const comm objects.

inline void **async_barrier**()

Asynchronous communicator barrier.

An async_barrier can match with other async_barrier and barrier calls. Any *comm::barrier()* calls matching with any async_barrier will execute as expected but will not return until all ranks are in a non-async barrier. The following code will complete successfully. If the calls to *async_barrier()* were replaced with *barrier()*, the code would deadlock with more than 1 rank. This call is useful when ranks may locally decide to run more iterations of a loop than other ranks.

```
for (int i=0; i<world.size(); ++i) {
  world.async_barrier();
}
world.barrier();
```

inline void **async_barrier**() const

Asynchronous communicator barrier that can be called on const comm objects.

inline void **local_progress**()

Checks for incoming unless called from receive queue and flushes one buffer.

inline bool **local_process_incoming**()

Check for incoming messages and continue processing until no messages are found.

Returns

True if any messages were received, otherwise false.

template<typename **Function**>
inline void **local_wait_until**(*Function* fn)

Waits until provided condition function returns true.

This is useful when applications can determine locally that their part of a computation is complete (or nearly complete). This can be used to completely avoid *barrier()* calls or reduce the number of reductions needed within a *barrier()* to reach quiescence.

```
  static int messages_received;
  messages_received = 0;
  for (int i=0; i<world.rank(); ++i) {
```

(continues on next page)

(continued from previous page)

```
        world.async(i, [](){++messages_received;});
    }

    world.local_wait_until([&world](){return messages_received ==
world.rank()});
```

Template Parameters

Function – functor type

Parameters

fn – Wait condition function, must match []() -> bool

template<typename **T**>
inline ygm_ptr<*T*> **make_ygm_ptr**(*T* &t)

inline void **register_pre_barrier_callback**(const std::function<void()> &fn)

Registers a callback that will be executed prior to the barrier completion.

Parameters

fn – callback function

template<typename **T**>
inline *T* **all_reduce_sum**(const *T* &t) const

Warning

Deprecated

template<typename **T**>
inline *T* **all_reduce_min**(const *T* &t) const

Warning

Deprecated

template<typename **T**>
inline *T* **all_reduce_max**(const *T* &t) const

Warning

Deprecated

template<typename **T**, typename **MergeFunction**>
inline *T* **all_reduce**(const *T* &t, *MergeFunction* merge) const

Warning

Deprecated

inline int `size()` const

Number of ranks in communicator.

Returns

Communicator size

inline int `rank()` const

Rank of the current process.

Ranks are unique IDs in the range [0, size-1] assigned to each process in the communicator.

Returns

Rank within communicator

inline MPI_Comm `get_mpi_comm()` const

Access to copy of underlying MPI communicator.

Returned MPI_Comm is still managed by YGM and will be freed during *ygm::comm* destructor.

Returns

Copy of MPI communicator distinct from one used for asynchronous communication

inline const detail::layout &`layout()` const

Access to underlying layout object.

Returns

ygm::detail::layout object used by the *ygm::comm*

inline const detail::comm_router &`router()` const

Access to underlying comm_router object.

Returns

ygm::detail::comm_router object used by the *ygm::comm*

inline bool `rank0()` const

Checks if current rank is rank 0.

Returns

bool indicating whether current rank is rank 0

template<typename `T`>
inline void `mpi_send`(const *T* &data, int dest, int tag, MPI_Comm comm) const

Send an MPI message.

Template Parameters

`T` – datatype being sent (must be serializable with cereal)

Parameters

- `data` – Message contents to send
- `dest` – Rank to send data to
- `tag` – MPI tag to assign to message
- `comm` – MPI communicator to send message over

template<typename `T`>

inline void **mpi_send**(const *T* &data, int dest, int tag) const

Send an MPI message over an unspecified MPI communicator.

Template Parameters

`T` – datatype being sent (must be serializable with cereal)

Parameters

- **`data`** – Message contents to send
- **`dest`** – Rank to send data to
- **`tag`** – MPI tag to assign to message

template<typename **T**>
inline *T* **mpi_recv**(int source, int tag, MPI_Comm comm) const

Receive an MPI message.

Template Parameters

`T` – datatype being received (must be serializable with cereal)

Parameters

- **`source`** – Rank sending message
- **`tag`** – MPI tag to assign to message
- **`comm`** – MPI communicator message is being sent over

Returns

Received message

template<typename **T**>
inline *T* **mpi_recv**(int source, int tag) const

Receive an MPI message over an unspecified MPI communicator.

Template Parameters

`T` – datatype being received (must be serializable with cereal)

Parameters

- **`source`** – Rank sending message
- **`tag`** – MPI tag to assign to message

Returns

Received message

template<typename **T**>
inline *T* **mpi_bcast**(const *T* &to_bcast, int root, MPI_Comm comm) const

Broadcast an MPI message.

Template Parameters

`Datatype` – to broadcast (must be serializable)

Parameters

- **`to_bcast`** – Data being broadcast
- **`root`** – Rank message is being broadcast from
- **`comm`** – MPI communicator message is being broadcast over

Returns

Data received from root

template<typename **T**>
inline *T* **mpi_bcast**(const *T* &to_bcast, int root) const

Broadcast an MPI message over an unspecified MPI communicator.

Template Parameters
`Datatype` – to broadcast (must be serializable)

Parameters

- **to_bcast** – Data being broadcast
- **root** – Rank message is being broadcast from

Returns
Data received from root

inline std::ostream &**cout0**() const

Provides a std::cout ostream that is only writeable from rank 0.

```
world.cout0() << "This output is coming from rank 0" << std::endl;
```

Returns
std::cout that only writes from rank 0

inline std::ostream &**cerr0**() const

Provides a std::cerr ostream that is only writeable from rank 0.

Returns
std::cerr that only writes from rank 0

inline std::ostream &**cout**() const

Provides std::cout access with each line labeled by the rank producing the output.

Returns
std::cout for use by any rank

inline std::ostream &**cerr**() const

Provides std::cerr access with each line labeled by the rank producing the output.

Returns
std::cerr for use by any rank

template<typename ...**Args**>
inline void **cout**(*Args*&&... args) const

python print-like function that writes to std::cout from any rank

```
world.cout("Printing from every rank")
```

Template Parameters
Args... – Variadic argument types to print

Parameters
args... – Variadic arguments for printing

template<typename ...**Args**>

inline void **cerr**(*Args*&&... args) const

python print-like function that writes to std::cerr from any rank

```
world.cerr("Printing from every rank")
```

Template Parameters

Args... – Variadic argument types to print

Parameters

args... – Variadic arguments for printing

template<typename ...**Args**>
inline void **cout0**(*Args*&&... args) const

python print-like function that writes to std::cout from only rank 0

```
world.cout0("Printing from rank 0 only")
```

Template Parameters

Args... – Variadic argument types to print

Parameters

args... – Variadic arguments for printing

template<typename ...**Args**>
inline void **cerr0**(*Args*&&... args) const

python print-like function that writes to std::cerr from only rank 0

Template Parameters

Args... – Variadic argument types to print

Parameters

args... – Variadic arguments for printing

inline void **enable_ygm_tracing**()

Turn on tracing of YGM functions.

This is more granular than MPI tracing. YGM tracing occurs at the level of individual async calls and is indicative of the calls requested by an application. MPI tracing occurs at the level of buffers sent through YGM and is indicative of the communication YGM actually performed to meet the requests of the application's async calls.

inline void **disable_ygm_tracing**()

Turn off tracing of YGM functions.

inline void **enable_mpi_tracing**()

Turn on tracing of MPI calls within YGM.

inline void **disable_mpi_tracing**()

Turn off tracing of MPI calls.

inline bool **is_ygm_tracing_enabled**() const

Check status of YGM tracing.

Returns

True if currently tracing YGM functions, otherwise false

inline bool **`is_mpi_tracing_enabled`**`()` const

Check status of MPI tracing.

Returns

True if currently tracing MPI calls, otherwise false

inline void **`set_log_level`**`(`const ygm::log_level level`)`

Set the log level to use in YGM.

Parameters

`level` – Log level to use. Possible values in order of increasing verbosity are ygm::log_level::off, ygm::log_level::critical, ygm::log_level::error, ygm::log_level::warn, ygm::log_level::info, ygm::log_level::debug

template<typename ...**`Args`**>
inline void **`log`**`(`const ygm::log_level level, *Args*&&... args`)` const

Add a message to the YGM logs.

```
int var = 6;
world.log(ygm::log_level::info, "This is my var: ", var);
```

Template Parameters

`Args...` – Variadic types to add to log

Parameters

`Minimum` – log level for logging message @args Variadic arguments add to log

template<typename **`StringType`**>
inline void **`set_log_location`**`(`const *StringType* &s`)`

Set the location of the YGM log files. One file will be created at this location for every rank.

Template Parameters

`StringType` – Type of provided path as string. Must be convertible to std::filesystem::path.

Parameters

`s` – Path to log location as a string

inline void **`set_log_location`**`(`std::filesystem::path p`)`

Set the location of the YGM log files. One file will be created at this location for every rank.

Parameters

`p` – Path to log location as an std::filesystem::path

Friends

friend class detail::interrupt_mask

friend class detail::comm_stats

struct **header_t**

Public Members

uint32_t **message_size**

int32_t **dest**

CHAPTER THREE

YGM::CONTAINER MODULE REFERENCE.

ygm::container is a collection of distributed containers designed specifically to perform well within YGM's asynchronous runtime. Inspired by C++'s Standard Template Library (STL), the containers provide improved programmability by allowing developers to consider an algorithm as the operations that need to be performed on the data stored in a container while abstracting the locality and access details of said data. While insiration is taken from STL, the top priority is to provide expressive and performant tools within the YGM framework.

3.1 Implemented Storage Containers

The currently implemented containers include a mix of distributed versions of familiar containers and distributed-specific containers:

- `ygm::container::bag` - An unordered collection of objects partitioned across processes. Ideally suited for iteration over all items with no capability for identifying or searching for an individual item within the bag.
- `ygm::container::set` - Analogous to `std::set`. An unordered collection of unique objects with the ability to iterate and search for individual items. Insertion and iteration are slower than a `ygm::container::bag`.
- `ygm::container::map` - Analogous to `std::map`. A collection of keys with assigned values. Keys and values can be inserted and looked up individually or iterated over collectively.
- `ygm::container::array` - A collection of items indexed by an integer type. Items can be inserted and looked up by their index values independently or iterated over collectively. Differs from a `std::array` in that sizes do not need to known at compile-time, and a `ygm::container::array` can be dynamically resized through a (potentially expensive) function at runtime.
- `ygm::container::counting_set` - A container for counting occurrences of items. Can be thought of as a `ygm::container::map` that maps items to integer counts but optimized for the case of frequent duplication of keys.
- `ygm::container::disjoint_set` - A distributed disjoint set data structure. Implements asynchronous union operation for maintaining membership of items within mathematical disjoint sets. Eschews the find operation of most disjoint set data structures and instead allows for execution of user-provided lambdas upon successful completion of set merges.

3.2 Typical Container Operations

Most interaction with containers occurs in one of two classes of operations: iteration and `async_`.

3.2.1 Iterating over Containers

Elements within a container can be iterated over using calls to `for_all` methods or using standard C++ iterators. In their standard form, both iteration techniques make calls to a YGM `barrier` on the underlying communicator to ensure that all updates to the container have been processed before starting the iteration. `local_` variants for both exist that skip the call to `barrier`, allowing them to be called in a non-collective context.

Container `for_all` Methods

`for_all`-class operations are barrier-inducing collectives that direct ranks to iteratively apply a user-provided function to all locally-held data. Functions passed to the `for_all` interface do not support additional variadic parameters. However, these functions are stored and executed locally on each rank, and so can capture objects in rank-local scope. The `local_for_all` variant has the same API as `for_all`, but skips the internal call to `barrier` at its beginning.

The following example shows a *for_all* being used to double all values in a `ygm::container::bag<int>` called `my_bag`:

```
int multiple{2};

my_bag.for_all([&multiple](int &value) {
  value = value * multiple;
  });
```

The above example uses a capture of the `multiple` variable that can be used within the lambda executed on each value within the bag.

Container Iterators

Iteration can also be performed using iterators. The `begin()` and `end()` methods return iterators to the local data stored within a rank. This allows for range-based for loops that have more control over the flow of the loop. For instance, this example adds all values within a `ygm::container::bag<int>` named `my_bag` until the first odd value is encountered:

```
int even_sum{0};

for (const auto &value : my_bag) {
  if (value % 2 == 1) {
    break;
  }

  even_sum += value;
}
```

When using iterators to YGM containers, it is important to remember that `begin()` and `end()` are collective calls that include a `barrier` to make sure all updates to the container have been processed. This can easily lead to deadlocks if not used carefully. The `local_begin()` and `local_end()` calls return the same iterators to the data within a rank as `begin()` and `end()` but do not call `barrier` at the beginning. These can be used to iterate locally within a single rank with the understanding that there may be messages queued that attempt to update values within the container which may need to be considered.

3.2.2 `async_` Operations

Operations prefixed with `async_` perform operations on containers that can be spawned from any process and execute on the correct process using YGM's asynchronous runtime. The most common `async` operations are:

- `async_insert` - Inserts an item or a key and value, depending on the container being used. The process responsible for storing the inserted object is determined using the container's partitioner. Depending on the container, this partitioner may determine this location using a hash of the item or by heuristics that attempt to evenly spread data across processes (in the case of `ygm::container::bag`).
- `async_visit` - Items within YGM containers will be distributed across the universe of running processes. Instead of providing operations to look up this data directly, which would involve a round-trip communication with the process storing the item of interest, most YGM containers provide `async_visit`. A call to `async_visit` takes a function to execute and arguments to pass to the function and asynchronously executes the provided function with arguments that are the item stored in the container and the additional arguments passed to `async_visit`.

Specific containers may have additional `async_` operations (or may be missing some of the above) based on the capabilities of the container. Consult the documentation of individual containers for more details.

3.3 YGM Container Example

```cpp
#include <ygm/comm.hpp>
#include <ygm/container/map.hpp>

int main(int argc, char **argv) {
  ygm::comm world(&argc, &argv);

  ygm::container::map<std::string, std::string> my_map(world);

  if (world.rank0()) {
    my_map.async_insert("dog", "bark");
    my_map.async_insert("cat", "meow");
  }

  world.barrier();

  auto favorites_lambda = [](auto key, auto &value, const int favorite_num) {
    std::cout << "My favorite animal is a " << key << ". It says '" << value
              << "!' My favorite number is " << favorite_num << std::endl;
  };

  // Send visitors to map
  if (world.rank() % 2) {
    my_map.async_visit("dog", favorites_lambda, world.rank());
  } else {
    my_map.async_visit("cat", favorites_lambda, world.rank() + 1000);
  }

```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

3.4 Container Transformation Objects

`ygm::container` provides a number of transformation objects that can be applied to containers to alter the appearance of items passed to `for_all` operations without modifying the items within the container itself. The currently supported transformation objects are:

- `filter` - Filters items in a container to only execute on the portion of the container satisfying a provided boolean function.
- `flatten` - Extract the elements from tuple-like objects before passing to the user's `for_all` function.
- `map` - Apply a generic function to the container's items before passing to the user's `for_all` function.

3.5 Container Class Documentation

3.5.1 array

template<typename **Value**, typename **Index** = size_t>

class **array** : public ygm::container::detail::base_async_insert_key_value<*array*<*Value*, size_t>, std::tuple<size_t, *Value*>>, public ygm::container::detail::base_misc<*array*<*Value*, size_t>, std::tuple<size_t, *Value*>>, public ygm::container::detail::base_async_visit<*array*<*Value*, size_t>, std::tuple<size_t, *Value*>>, public ygm::container::detail::base_iteration_key_value<*array*<*Value*, size_t>, std::tuple<size_t, *Value*>>, public ygm::container::detail::base_async_reduce<*array*<*Value*, size_t>, std::tuple<size_t, *Value*>>

Container for key-value pairs with keys that are contiguous indices in the range [0, *size()*-1].

Assigns ranks contiguous chunks of indices using block_partitioner object. Resizing array is an expensive operation as it requires reassigning storage to ranks.

Public Types

using **self_type** = *array*<*Value*, *Index*>

using **mapped_type** = *Value*

using **key_type** = *Index*

using **size_type** = *Index*

using **for_all_args** = std::tuple<*Index*, *Value*>

using **container_type** = ygm::container::array_tag using **ptr_type** = typename ygm::ygm_ptr<*self_type*>

Public Functions

array() = delete

inline **array**(ygm::*comm* &comm, const *size_type* size)

Array constructor.

Parameters

- **comm** – Communicator to use for communication
- **size** – Global size to use to array

inline **array**(ygm::*comm* &comm, const *size_type* size, const *mapped_type* &default_value)

Array constructor taking default value.

Parameters

- **comm** – Communicator to use for communication
- **size** – Global size to use for array
- **default_value** – Value to initialize all stored items with

inline **array**(ygm::*comm* &comm, std::initializer_list<*mapped_type*> l)

Array constructor from std::initializer_list of values.

Initializer list is assumed to be replicated on all ranks. Initializer list only contains values to place in array. Indices assigned to values are provided in sequential order. Array size is determined by size of initializer list.

Parameters

- **comm** – Communicator to use for communication
- **l** – Initializer list of values to put in array

inline **array**(ygm::*comm* &comm, std::initializer_list<std::tuple<*key_type*, *mapped_type*>> l)

Array constructor from std::initializer_list of index-value pairs.

Initializer list is assumed to be replicated on all ranks. Initializer list contains index-value pairs to place in array. Indices are not assumed to be in sequential order or contiguous. Array size is determined by max index within initializer list.

Parameters

- **comm** – Communicator to use for communication
- **l** – Initializer list of index-value pairs to put in array

template<typename **T**>
inline **array**(ygm::*comm* &comm, const *T* &t)

requires detail

Construct array from existing YGM container.

Existing container contains only values. Indices are assigned sequentially across ranks. Partitioning will likely not be the same between existing container and constructed array.

Template Parameters

`T` – Existing container type

Parameters

- **`comm`** – Communicator to use for communication
- **`t`** – YGM container containing values to put in array

template<typename **T**>
inline **`array`**(ygm::*comm* &comm, const *T* &t)
requires detail

Construct array from existing YGM container of key-value pairs.

Requires input container `for_all_args` to be a single item tuple that is itself a key-value pair (e.g. works from a *`ygm::container::bag`*). Array size is determined by finding the largest index across all ranks.

Template Parameters

`T` – Existing container type

Parameters

- **`comm`** – Communicator to use for communication
- **`t`** – YGM container of key-value pairs to put in array.

template<typename **T**>
inline **`array`**(ygm::*comm* &comm, const *T* &t)
requires detail

Construct array from existing YGM container of key-value pairs.

Requires input container's `for_all_args` to be a tuple containing keys and values (e.g. works from a *`ygm::container::map`*). Array size is determined by finding the largest index across all ranks.

Template Parameters

`T` – Existing container type

Parameters

- **`comm`** – Communicator to use for communication
- **`t`** – YGM container of key-value pairs to put in array

template<typename **T**>
inline **`array`**(ygm::*comm* &comm, const *T* &t)
requires detail

Construct array from existing STL container.

Existing container contains only values. Values are assumed to be distinct between ranks. Indices are assigned sequentially across ranks. Partitioning will likely not be the same between existing container and constructed array.

Template Parameters
`T` – Existing container type

Parameters

- `comm` – Communicator to use for communication
- `t` – STL container containing values to put in array

template<typename `T`>
inline `array`(ygm::*comm* &comm, const *T* &t)
requires detail

Construct array from existing STL container of key-value pairs.

Requires existing container to have a `value_type` that contains keys and values. Array size is determined by finding the largest index across all ranks.

Template Parameters
`T` – Existing container type

Parameters

- `comm` – Communicator to use for communication
- `t` – STL container of key-value pairs to put in array.

inline `~array`()

inline `array`(const *self_type* &other)

inline `array`(*self_type* &&other) noexcept

inline *array* &`operator=`(const *self_type* &other)

inline *array* &`operator=`(*self_type* &&other) noexcept

inline void `local_insert`(const *key_type* &key, const *mapped_type* &value)

Insert a key and value into local storage.

Assumes key (index) has already been converted to a local index.

Parameters

- `key` – Local index to store value at
- `value` – Vale to store

template<typename `Function`, typename ...`VisitorArgs`>
inline void `local_visit`(const *key_type* index, *Function* &&fn, const *VisitorArgs*&... args)

Visit an item stored locally.

Template Parameters

- `Function` – functor type
- `VisitorArgs...` – Variadic argument types

Parameters

- `index` – Index to visit
- `fn` – User-provided function to execute at item

- **`args...`** – Arguments to pass to user functor

inline void **`async_set`**(const *key_type* index, const *mapped_type* &value)

Set the value associated to given index.

Parameters

- **`index`** – Index to store value at
- **`value`** – Value to store

template<typename **`BinaryOp`**>
inline void **`async_binary_op_update_value`**(const *key_type* index, const *mapped_type* &value, const *BinaryOp* &b)

Apply a binary operation to a provided value and the value already stored at a given index to update the stored value.

Template Parameters

`BinaryOp` – functor type

Parameters

- **`index`** – Index to apply update at
- **`value`** – New value to update with
- **`b`** – Binary operation to apply

inline void **`async_bit_and`**(const *key_type* index, const *mapped_type* &value)

Apply bitwise and to update stored value.

Parameters

- **`index`** – Index to perform update at
- **`value`** – Value to "and" with current value

inline void **`async_bit_or`**(const *key_type* index, const *mapped_type* &value)

Apply bitwise or to update stored value.

Parameters

- **`index`** – Index to perform update at
- **`value`** – Value to "or" with current value

inline void **`async_bit_xor`**(const *key_type* index, const *mapped_type* &value)

Apply bitwise xor to update stored value.

Parameters

- **`index`** – Index to perform update at
- **`value`** – Value to "xor" with current value

inline void **`async_logical_and`**(const *key_type* index, const *mapped_type* &value)

Apply logical and to update stored value.

Parameters

- **`index`** – Index to perform update at
- **`value`** – Value to "and" with current value

inline void **async_logical_or**(const *key_type* index, const *mapped_type* &value)

Apply logical or to update stored value.

Parameters

- **index** – Index to perform update at
- **value** – Value to “or” with current value

inline void **async_multiplies**(const *key_type* index, const *mapped_type* &value)

Apply multiplication to update stored value.

Parameters

- **index** – Index to perform update at
- **value** – Value to multiply with current value

inline void **async_divides**(const *key_type* index, const *mapped_type* &value)

Apply division to update stored value.

Parameters

- **index** – Index to perform update at
- **value** – Value to divide current value by

inline void **async_plus**(const *key_type* index, const *mapped_type* &value)

Apply addition to update stored value.

Parameters

- **index** – Index to perform update at
- **value** – Value to add to current value

inline void **async_minus**(const *key_type* index, const *mapped_type* &value)

Apply subtraction to update stored value.

Parameters

- **index** – Index to perform update at
- **value** – Value to subtract from current value

template<typename **UnaryOp**>
inline void **async_unary_op_update_value**(const *key_type* index, const *UnaryOp* &u)

Apply a unary operation to the value already stored at a given index to update the stored value.

Template Parameters

UnaryOp – functor type

Parameters

- **index** – Index to apply update at
- **u** – Unary operation to apply

inline void **async_increment**(const *key_type* index)

Increment stored value.

Parameters

index – Index to perform update at

inline void **`async_decrement`**(const *key_type* index)

Decrement stored value.

Parameters
`index` – Index to perform update at

const *mapped_type* &**`default_value`**() const

inline void **`resize`**(const *size_type* size, const *mapped_type* &fill_value)

Set new global size for array.

This operation requires repartitioning the data already stored in a container, which is a `O(old_size)` operation.

Parameters

- **`size`** – New global size
- **`fill_value`** – Value to initialize new values to (when expanding an array)

inline void **`resize`**(const *size_type* size)

Set new global size for array with a default fill value.

Equivalent to `resize(size, m_default_value)`

Parameters
`size` – New global size

inline size_t **`local_size`**()

Get the number of elements stored on the local process.

Returns
Local size of array

inline size_t **`size`**() const

Get the global size of the array.

Returns
Array's global size

inline void **`local_clear`**()

Clear the local contents of the array and set size to 0.

Setting the local size to 0 cannot be performed independently of other ranks. This operation needs to be called collectively for the array.

inline void **`local_swap`**(*self_type* &other)

Swap the local contents of an array.

Parameters
`other` – The array to swap local contents with

template<typename **`Function`**>
inline void **`local_for_all`**(*Function* &&fn)

Apply a lambda to all local elements.

This operation can be called non-collectively.

Template Parameters
`Function` – functor type

Parameters
`fn` – Functor object to apply to all elements locally stored in the array

template<typename **ReductionOp**>
inline void **local_reduce**(const *key_type* index, const *mapped_type* &value, *ReductionOp* reducer)

Update a locally stored element by performing a binary operation between it and a provided value.

Template Parameters
`ReductionOp` – functor type

Parameters

- `index` – Global index to perform binary operation at. Must be found on the local process.
- `value` – Value to combine with the currently-held value
- `reducer` – Binary operation to perform

inline void **sort**()

Globally sort values in array in increasing order.

Partitions data using sampled pivots to approximately balance values on ranks. Then use `std::sort` locally on values before reinserting into the array.

inline void **async_insert**(const typename std::tuple_element<0, std::tuple<size_t, *Value*>>::type &key, const typename std::tuple_element<1, std::tuple<size_t, *Value*>>::type &value)
requires DoubleItemTuple<std::tuple<size_t, *Value*>>

Asynchronously insert a key-value pair into a container.

The container's local_insert() function is free to determine the behavior when `key` is already in the container

```
ygm::container::map<int, std::string> my_map(world);
my_map.async_insert(1, "one");
```

Parameters

- `key` – Key to insert
- `value` – Value to associate to key

inline void **async_insert**(const std::pair<const typename std::tuple_element<0, std::tuple<size_t, *Value*>>::type, typename std::tuple_element<1, std::tuple<size_t, *Value*>>::type> &kvp)
requires DoubleItemTuple<std::tuple<size_t, *Value*>>

Asynchronously insert a key-value pair into a container.

Equivalent to `async_insert(kvp.first, kvp.second)`

Parameters
`kvp` – Key-value pair to insert

inline void **clear**()

Clears the contents of a YGM container.

inline void **swap**(*array*<*Value*, size_t> &other)

Swaps the contents of a YGM container.

Parameters

other – Container to swap with

inline ygm::*comm* &**comm**() const

Access to underlying YGM communicator.

Returns

YGM communicator used for communication by container

inline ygm::ygm_ptr<*array*<*Value*, size_t>> **get_ygm_ptr**()

Access to the ygm_ptr used by the container.

Returns

`ygm_ptr` used by the container when identifying itself in `async` calls on the *`ygm::comm`*

inline const ygm::ygm_ptr<*array*<*Value*, size_t>> **get_ygm_ptr**() const

Const access to the ygm ptr used by the container.

Returns

`ygm_ptr` to const version of container

inline void **async_visit**(const typename std::tuple_element<0, std::tuple<size_t, *Value*>>::type &key, Visitor &&visitor, const VisitorArgs&... args)
requires DoubleItemTuple<std::tuple<size_t, *Value*>>

Asynchronously visit key within a container and execute a user-provided function.

```
ygm::container::map<std::string, int> my_map(world);
my_map.async_insert("one", 1);
world.barrier();
my_map.async_visit("one", [](const auto &key, auto &val, int &to_add) {
   val += to_add;
   }, world.size())
world.barrier();
```

will result in a value of `world.size() * world.size() + 1` associated to the key `"one"`.

Template Parameters

- **Visitor** – Type of user-provided function
- **VisitorArgs...** – Variadic argument types to give to user function

Parameters

- **key** – Key to visit in container
- **visitor** – User-provided function to execute at key
- **args...** – Variadic arguments to pass to user-provided function

inline void **async_visit_if_contains**(const typename std::tuple_element<0, std::tuple<size_t, *Value*>>::type &key, Visitor visitor, const VisitorArgs&... args)
requires DoubleItemTuple<std::tuple<size_t, *Value*>>

Asynchronously visit key within a container and execute a user-provided function only if the key already exists.

This function differs from `async_visit` in that it will not default construct a value within the container prior to visiting a key that does not already exist.

Template Parameters

- **`Visitor`** – Type of user-provided function
- **`VisitorArgs...`** – Variadic argument types to give to user function

Parameters

- **`key`** – Key to visit in container
- **`visitor`** – User-provided function to execute at key
- **`args...`** – Variadic arguments to pass to user-provided function

inline void **`async_visit_if_contains`**(const typename std::tuple_element<0, std::tuple<size_t, *Value*>>::type &key, Visitor visitor, const VisitorArgs&... args) const

requires DoubleItemTuple<std::tuple<size_t, *Value*>>

Version of `async_visit_if_contains` that works on `const` objects and provides `const` arguments to the user-provided lambda.

inline void **`for_all`**(Function fn)

Iterates over all items in a container and executes a user-provided function object on each.

The user-provided function is expected to take a single key and value as separate arguments that make a (key, value) pair within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters

`Function` – Type of user-provided function

Parameters

`fn` – User-provided function

inline void **`for_all`**(Function &&fn) const

Const version of for_all that iterates over all items and passes them to the user function as const *.

The user-provided function is expected to take a single key and value as separate arguments that make a (key, value) pair within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters

`Function` – Type of user-provided function

Parameters

`fn` – User-provided function

inline void **`gather`**(STLContainer &gto, int rank) const

Gather all values in an STL container.

Requires STL container to have a `value_type` that is (key, value) pairs from the YGM container

Template Parameters

`STLContainer` – Type of STL container to gather to

Parameters

- **gto** – Container to store results in
- **rank** – Rank to tather results on. Use -1 to gather to all ranks

inline void **gather**(STLContainer &gto) const

Gather all values in an STL container on all ranks.

Equivalent to `gather(gto, -1)`

Template Parameters

STLContainer – Type of STL container to gather to

Parameters

gto – Container to store results in

inline std::vector<std::pair<*key_type*, *mapped_type*>> **gather_topk**(size_t k, Compare comp = Compare()) const

Gather the k “largest” key-value pairs according to provided comparison function.

Template Parameters

Compare – Type of comparison operator

Parameters

- **k** – Number of key-value pairs to gather
- **comp** – Comparison function for identifying elements to gather

Returns

vector of largest key-value pairs

inline void **collect**(YGMContainer &c) const

Collects all items in a new YGM container.

Template Parameters

YGMContainer – Container type

Parameters

c – Container to collect into

inline void **reduce_by_key**(MapType &map, ReductionOp reducer) const

Reduces all values in key-value pairs with matching keys.

Template Parameters

- **MapType** – Result YGM container type
- **ReductionOp** – Functor type

Parameters

- **map** – YGM container to hold result
- **reducer** – Functor for combining values

transform_proxy_key_value<*array*<*Value*, size_t>, TransformFunction> **transform**(TransformFunction &&ffn)

Creates proxy that transforms key-value pairs in a container that are presented to user `for_all` calls.

The underlying items within the container are not modified.

Template Parameters
`TransformFunction` – functor type

Parameters
`ffn` – Function to transform items in container

inline auto **`keys`**()

Access to container presenting only keys.

Returns
Transform object that returns only keys to user

inline auto **`values`**()

Access to container presenting only values.

Returns
Transform object that returns only values to user

inline flatten_proxy_key_value<*array*<*Value*, size_t>> **`flatten`**()

Flattens STL containers of values to allow a function to be called on inner items individually.

Underlying container is not modified.

filter_proxy_key_value<*array*<*Value*, size_t>, FilterFunction> **`filter`**(FilterFunction &&ffn)

Filters items in a container so only allow `for_all` to execute on those that satisfy a given predicate function.

Filtered items are not removed from underlying container.

Template Parameters
`FilterFunction` – Functor type

Parameters
`ffn` – Function used to filter items in container.

inline void **`async_reduce`**(const typename std::tuple_element<0, std::tuple<size_t, *Value*>>::type &key, const typename std::tuple_element<1, std::tuple<size_t, *Value*>>::type &value, ReductionOp reducer)

Combines existing `mapped_type` item with `value` using a user-provided binary operation if `key` is found in container. Inserts default `mapped_type` prior to reduction if `key` does not already exist in container.

```
ygm::container::map<std::string, int> my_map(world);
my_map.async_insert("one", 1);
if (world.rank0()) {
   my_map.async_reduce("one", 2, std::plus<int>());
   my_map.async_reduce("two", 2, std::plus<int>());
}
world.barrier()
```

will result in `my_map` containing the pairs `("one", 3)` and `("two", 2)`.

Template Parameters
`ReductionOp` – Type of function provided by usert to perform reduction

Parameters

- **`key`** – Key to search for within container
- **`value`** – Provided value to combine with existing value in container

Public Members

detail::block_partitioner<*key_type*> **partitioner**

Friends

friend class detail::base_misc< array< Value, Index >, std::tuple< Index, Value > >

3.5.2 bag

template<typename **Item**>

class **bag** : public ygm::container::detail::base_async_insert_value<*bag*<*Item*>, std::tuple<*Item*>>, public ygm::container::detail::base_count<*bag*<*Item*>, std::tuple<*Item*>>, public ygm::container::detail::base_misc<*bag*<*Item*>, std::tuple<*Item*>>, public ygm::container::detail::base_iterators<*bag*<*Item*>>, public ygm::container::detail::base_iteration_value<*bag*<*Item*>, std::tuple<*Item*>>

Container that partitions elements across ranks for iteration.

Assigns items in a cyclic distribution from every rank independently

Public Types

using **self_type** = *bag*<*Item*>

using **value_type** = *Item*

using **size_type** = size_t

using **for_all_args** = std::tuple<*Item*>

using **container_type** = ygm::container::bag_tag

using **iterator** = typename local_container_type::iterator

using **const_iterator** = typename local_container_type::const_iterator

Public Functions

inline **bag**(ygm::*comm* &comm)

Bag construction from *ygm::comm*.

Parameters

`comm` – Communicator to use for communication

inline **bag**(ygm::*comm* &comm, std::initializer_list<*Item*> l)

Bag constructor from std::initializer_list of values.

Initializer list is assumed to be replicated on all ranks.

Parameters

- `comm` – Communicator to use for communication
- `l` – Initializer list of values to put in bag

template<typename **STLContainer**>
inline **bag**(ygm::*comm* &comm, const *STLContainer* &cont)
requires detail

Construct bag from existing STL container.

Template Parameters

`STLContainer` – Existing container type

Parameters

- `comm` – Communicator to use for communication
- `cont` – STL container containing values to put in bag

template<typename **YGMContainer**>
inline **bag**(ygm::*comm* &comm, const *YGMContainer* &yc)
requires detail

Construct bag from existing YGM container.

Requires container's `for_all_args` to be a single item tuple to put in the bag

Template Parameters

`YGMContainer` – Existing container type

Parameters

- `comm` – Communicator to use for communication
- `yc` – YGM container of values to put in bag

inline **~bag**()

inline **bag**(const *self_type* &other)

inline **bag**(*self_type* &&other) noexcept

inline *bag* &**operator=**(const *self_type* &other)

inline *bag* &**operator=**(*self_type* &&other) noexcept inline *iterator* **`local_begin`**`()`

Access to begin iterator of locally-held items.

Does not call `barrier()`.

Returns

Local iterator to beginning of items held by process.

inline *const_iterator* **`local_begin`**`()` const

Access to begin const_iterator of locally-held items for const bag.

Does not call `barrier()`.

Returns

Local const iterator to beginning of items held by process.

inline *const_iterator* **`local_cbegin`**`()` const

Access to begin const_iterator of locally-held items for const bag.

Does not call `barrier()`.

Returns

Local const iterator to beginning of items held by process.

inline *iterator* **`local_end`**`()`

Access to end iterator of locally-held items.

Does not call `barrier()`.

Returns

Local iterator to end of items held by process.

inline *const_iterator* **`local_end`**`()` const

Access to end const_iterator of locally-held items for const bag.

Does not call `barrier()`.

Returns

Local const iterator to ending of items held by process.

inline *const_iterator* **`local_cend`**`()` const

Access to end const_iterator of locally-held items for const bag.

Does not call `barrier()`.

Returns

Local const iterator to ending of items held by process.

inline void **`async_insert`**`(`const *Item* &value, int dest`)`

Asynchronously insert an item on a specific rank.

Parameters

- `value` – Value to insert in bag
- `dest` – Rank to insert item at

inline void **`async_insert`**(const std::vector<*Item*> &values, int dest)

Asynchronously insert multiple items on a specific rank.

Parameters

- `values` – Vector of values to insert in bag
- `dest` – Rank to insert items at

inline void **`local_insert`**(const *Item* &val)

Insert an item into local storage.

Parameters
`val` – Value to insert locally

inline void **`local_clear`**()

Clear the local storage of the bag.

inline size_t **`local_size`**() const

Get the number of items held locally.

Returns
Number of locally-held items

inline size_t **`local_count`**(const *value_type* &val) const

Count the number of items held locally that match a query item.

Parameters
`val` – Value to search for locally

Returns
Number of locally-held copies of val

template<typename **`Function`**>
inline void **`local_for_all`**(*Function* &&fn)

Execute a functor on every locally-held item.

Template Parameters
`Function` – functor type

Parameters
`fn` – Functor to execute on items

template<typename **`Function`**>
inline void **`local_for_all`**(*Function* &&fn) const

Execute a functor on a `const` version of every locally-held item.

Template Parameters
`Function` – functor type

Parameters
`fn` – Functor to execute on items

inline void **`serialize`**(const std::string &fname)

Serialize a bag to a collection of files to be read back in later.

Parameters
`fname` – Filename prefix to create filename used by every rank from

inline void **`deserialize`**(const std::string &fname)

Deserialize a bag from files.

Currently requires the number of ranks deserializing a bag to be the same as was used for serialization.

Parameters

`fname` – Filename prefix to create filename used by every rank from

inline void **`rebalance`**()

Repartition data to hold approximately equal numbers of items on every rank.

template<typename **`RandomFunc`**>
inline void **`local_shuffle`**(*RandomFunc* &r)

Shuffle elements held locally.

Template Parameters

`RandomFunc` – Random number generator type

Parameters

`r` – Random number generator

inline void **`local_shuffle`**()

Shuffle elements held locally with a default random number generator.

template<typename **`RandomFunc`**>
inline void **`global_shuffle`**(*RandomFunc* &r)

Shuffle elements of bag across ranks.

Template Parameters

`RandomFunc` – Random number generator type

Parameters

`r` – Random number generator

inline void **`global_shuffle`**()

Shuffle elements of bag across ranks with a default random number generator.

inline void **`async_insert`**(const typename std::tuple_element<0, std::tuple<*Item*>>::type &value)
requires SingleItemTuple<std::tuple<*Item*>>

Asynchronously inserts a value in a container.

```
ygm::container::bag<int> my_bag(world);
my_bag.async_insert(world.rank());
```

Parameters

`value` – Value to insert into container

inline size_t **`count`**(const typename std::tuple_element<0, std::tuple<*Item*>>::type &value) const

Counts all occurrences of a value within a container.

Parameters

`value` – Value to search for within container (key in the case of containers with keys)

Returns

Count of times `value` is seen in container inline size_t **size**() const

Gets number of elements in a YGM container.

Returns

Container size

inline void **clear**()

Clears the contents of a YGM container.

inline void **swap**(*bag*<*Item*> &other)

Swaps the contents of a YGM container.

Parameters

other – Container to swap with

inline ygm::*comm* &**comm**() const

Access to underlying YGM communicator.

Returns

YGM communicator used for communication by container

inline ygm::ygm_ptr<*bag*<*Item*>> **get_ygm_ptr**()

Access to the ygm_ptr used by the container.

Returns

`ygm_ptr` used by the container when identifying itself in `async` calls on the *`ygm::comm`*

inline const ygm::ygm_ptr<*bag*<*Item*>> **get_ygm_ptr**() const

Const access to the ygm ptr used by the container.

Returns

`ygm_ptr` to const version of container

inline auto **begin**()

Returns an iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The iterator is a local iterator, not a global iterator

Returns

iterator to the beginning of the local container's data.

inline auto **begin**() const

Returns a const_iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the beginning of the local container's data.

inline auto **`cbegin`**`()` const

Returns a const_iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the beginning of the local container's data.

inline auto **`end`**`()`

Returns an iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The iterator is a local iterator, not a global iterator

Returns

iterator to the end of the local container's data.

inline auto **`end`**`()` const

Returns a const_iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the end of the local container's data.

inline auto **`cend`**`()` const

Returns a const_iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the end of the local container's data.

inline void **`for_all`**`(`Function &&fn`)`

Iterates over all items in a container and executes a user-provided function object on each.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters

`Function` – Type of user-provided function

Parameters

`fn` – User-provided function

inline void **`for_all`**`(`Function &&fn`)` const

Const version of for_all that iterates over all items and passes them to the user function as const *.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters

`Function` – Type of user-provided function

Parameters

`fn` – User-provided function

inline void **`gather`**`(`STLContainer &gto, int rank`)` const

Gather all values in an STL container.

Template Parameters

`STLContainer` – Type of STL container to gather to

Parameters

- **`gto`** – Container to store results in
- **`rank`** – Rank to tather results on. Use -1 to gather to all ranks

inline void **`gather`**`(`STLContainer &gto`)` const

Gather all values in an STL container on all ranks.

Equivalent to `gather(gto, -1)`

Template Parameters

`STLContainer` – Type of STL container to gather to

Parameters

`gto` – Container to store results in

inline std::vector<*value_type*> **`gather_topk`**`(`size_t k, Compare comp = std::greater<*value_type*>()`)` const requires SingleItemTuple<*for_all_args*>

Gather the k "largest" values according to provided comparison function.

Template Parameters

`Compare` – Type of comparison operator

Parameters

- `k` – Number of values to gather
- `comp` – Comparison function for identifying elements to gather

Returns

vector of largest values

inline *value_type* `reduce`(MergeFunction merge) const

Perform a reduction over all items in container.

`reduce` only makes sense to use with commutative and associative functors defining merges. Otherwise, ranks will not receive the same result.

Template Parameters

`MergeFunction` – Merge functor type

Parameters

`merge` – Functor to combine pairs of items

Returns

Value from all reductions

inline void `collect`(YGMContainer &c) const

Collects all items in a new YGM container.

Template Parameters

`YGMContainer` – Container type

Parameters

`c` – Container to collect into

inline void `reduce_by_key`(MapType &map, ReductionOp reducer) const

Reduces all values in key-value pairs with matching keys.

Template Parameters

- `MapType` – Result YGM container type
- `ReductionOp` – Functor type

Parameters

- `map` – YGM container to hold result
- `reducer` – Functor for combining values

transform_proxy_value<*bag*<*Item*>, TransformFunction> `transform`(TransformFunction &&ffn)

Creates proxy that transforms items in container that are presented to user `for_all` calls.

The underlying items within the container are not modified.

```
ygm::container::bag<int> my_bag(world);
my_bag.async_insert(2);
my_bag.barrier();

my_bag.transform([](auto &val) { return 2*val; }).for_all([](const auto
&transformed_val) { YGM_ASSERT_RELEASE(val == 4);
});

my_bag.for_all([](const auto &val) { YGM_ASSERT_RELEASE(val == 2); });
```

will complete successfully.

Template Parameters
`TransformFunction` – functor type

Parameters
`ffn` – Function to transform items in container

inline flatten_proxy_value<*bag*<*Item*>> **`flatten`**()

Flattens STL containers of values to allow a function to be called on inner items individually.

Underlying container is not modified.

```
ygm::container::bag<std::vector<int>> my_bag(world, {{1, 2, 3}});

my_bag.flatten().for_all([](const int &nested_val) {
std::cout << "Nested value: " << nested_val << std::cout;
});
```

will print

```
Nested value: 1
Nested value: 2
Nested value: 3
```

filter_proxy_value<*bag*<*Item*>, FilterFunction> **`filter`**(FilterFunction &&ffn)

Filters items in a container so only allow `for_all` to execute on those that satisfy a given predicate function.

Filtered items are not removed from underlying container.

```
ygm::container::bag<int> my_bag(world, {1, 2, 3, 4});
my_bag.filter([](const auto &val) { return (val % 2) == 0;
}).for_all([](const auto &filtered_val) { YGM_ASSERT_RELEASE((filtered_val %
2) == 0);
});
```

Template Parameters
`FilterFunction` – Functor type

Parameters
`ffn` – Function used to filter items in container.

Public Members

detail::round_robin_partitioner **partitioner**

Friends

friend class detail::base_misc< bag< Item >, std::tuple< Item > >

3.5.3 counting_set

template<typename **Key**>

class **counting_set** : public ygm::container::detail::base_count<*counting_set*<*Key*>, std::tuple<*Key*, size_t>>, public ygm::container::detail::base_misc<*counting_set*<*Key*>, std::tuple<*Key*, size_t>>, public ygm::container::detail::base_iterators<*counting_set*<*Key*>>, public ygm::container::detail::base_iteration_key_value<*counting_set*<*Key*>, std::tuple<*Key*, size_t>>

ygm::container::map that is specialized for counting occurrences of items in a stream.

Adds a local cache of objects to reduce sends of frequently-occurring items

Public Types

using **self_type** = *counting_set*<*Key*>

using **mapped_type** = size_t

using **key_type** = *Key*

using **size_type** = size_t

using **for_all_args** = std::tuple<*Key*, size_t>

using **container_type** = ygm::container::counting_set_tag

using **iterator** = typename internal_container_type::iterator

using **const_iterator** = typename internal_container_type::const_iterator

Public Functions

inline **counting_set**(ygm::*comm* &comm)

counting_set constructor

Parameters

comm – Communicator to use for communication

counting_set() = delete

inline **counting_set**(ygm::*comm* &comm, std::initializer_list<*Key*> l)

counting_set constructor from std::initializer_list of values

Initializer list is assumed to be replicated on all ranks.

Parameters

- **comm** – Communicator to use for communication
- **l** – Initializer list of values to put in *counting_set*

template<typename **STLContainer**>
inline **counting_set**(ygm::*comm* &comm, const *STLContainer* &cont)
requires detail

Construct *counting_set* by counting items in existing STL container.

Template Parameters

STLContainer – Existing container type

Parameters

- **comm** – Communicator to use for communication
- **cont** – STL container containing values to count

template<typename **YGMContainer**>
inline **counting_set**(ygm::*comm* &comm, const *YGMContainer* &yc)
requires detail

Construct *counting_set* by counting items in existing YGM container.

Template Parameters

YGMContainer – Existing container type

Parameters

- **comm** – Communicator to use for communication
- **yc** – YGM container containing values to count

inline **~counting_set**()

inline **counting_set**(const *self_type* &other)

inline **counting_set**(*self_type* &&other)

inline *counting_set* &**operator=**(const *self_type* &other)

inline *counting_set* &**operator=**(*self_type* &&other)

inline *iterator* **`local_begin`**`()`

Access to begin iterator of locally-held items.

Does not call `barrier()`.

Returns

Local iterator to beginning of items held by process.

inline *const_iterator* **`local_begin`**`()` const

Access to begin const_iterator of locally-held items for const *counting_set*.

Does not call `barrier()`.

Returns

Local const iterator to beginning of items held by process.

inline *const_iterator* **`local_cbegin`**`()` const

Access to begin const_iterator of locally-held items for const *counting_set*.

Does not call `barrier()`.

Returns

Local const iterator to beginning of items held by process.

inline *iterator* **`local_end`**`()`

Access to end iterator of locally-held items.

Does not call `barrier()`.

Returns

Local iterator to end of items held by process.

inline *const_iterator* **`local_end`**`()` const

Access to end const_iterator of locally-held items for const *counting_set*.

Does not call `barrier()`.

Returns

Local const iterator to ending of items held by process.

inline *const_iterator* **`local_cend`**`()` const

Access to end const_iterator of locally-held items for const *counting_set*.

Does not call `barrier()`.

Returns

Local const iterator to ending of items held by process.

inline void **`async_insert`**`(`const *key_type* &key`)`

Asynchronously insert an item for counting.

Inserts item into local cache before sending count to remote location

Parameters
`key` – Item to count

template<typename `Function`>
inline void `local_for_all`(*Function* &&fn)

Execute a functor on every locally-held item and count.

Template Parameters
`Function` – functor type

Parameters
`fn` – Functor to execute on items and counts

template<typename `Function`>
inline void `local_for_all`(*Function* &&fn) const

Execute a functor on every locally-held item and count for a const container.

Template Parameters
`Function` – functor type

Parameters
`fn` – Functor to execute on items and counts

inline void `local_clear`()

Clear the local storage of the *counting_set*.

inline void `clear`()

Clear the global storage of the *counting_set*.

inline size_t `local_size`() const

Get the number of items held locally.

Returns
Number of locally-held items

inline *mapped_type* `local_count`(const *key_type* &key) const

Get the total number of times a locally-held item has been counted so far.

Counts can be inaccurate before a `barrier()` due to items still being cached on other processes or waiting to be sent in *`ygm::comm`* buffers.

Returns
Number of times a locally-held item has been counted

inline *mapped_type* `count_all`()

Count the total number of items counted.

Returns
Sum of all item counts

template<typename `CompareFunction`>
inline std::vector<std::pair<*key_type*, *mapped_type*>> `topk`(size_t k, *CompareFunction* cfn)

Gather the k “largest” item-count pairs according to provided comparison function.

Template Parameters
`Compare` – Type of comparison operator

Parameters

- `k` – Number of item-count pairs to gather

- **comp** – Comparison function for identifying elements to gather

Returns

vector of largest item-count pairs

inline std::map<*key_type*, *mapped_type*> **gather_keys**(const std::vector<*key_type*> &keys)

Collective operation to look up item counts from each rank.

Parameters

keys – Keys local rank wants to collect counts for

Returns

`std::map` of provided keys and their counts

inline ygm::ygm_ptr<*self_type*> **get_ygm_ptr**() const

Access to the ygm_ptr used by the container.

Returns

`ygm_ptr` used by the container when identifying itself in `async` calls on the *`ygm::comm`*

inline void **serialize**(const std::string &fname)

Serialize counting set contents to collection of files.

Parameters

fname – Filename prefix to create names for files used by each rank

inline void **deserialize**(const std::string &fname)

Deserialize counting set contents from collection of files.

Parameters

fname – Filename prefix to create names for files used by each rank

inline size_t **count**(const typename std::tuple_element<0, std::tuple<*Key*, size_t>>::type &value) const

Counts all occurrences of a value within a container.

Parameters

value – Value to search for within container (key in the case of containers with keys)

Returns

Count of times `value` is seen in container

inline size_t **size**() const

Gets number of elements in a YGM container.

Returns

Container size

inline void **swap**(*counting_set*<*Key*> &other)

Swaps the contents of a YGM container.

Parameters

other – Container to swap with

inline ygm::*comm* &**comm**() const

Access to underlying YGM communicator.

Returns

YGM communicator used for communication by container inline ygm::ygm_ptr<*counting_set*<*Key*>> **`get_ygm_ptr`**`()`

Access to the ygm_ptr used by the container.

Returns

`ygm_ptr` used by the container when identifying itself in `async` calls on the *`ygm::comm`*

inline auto **`begin`**`()`

Returns an iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The iterator is a local iterator, not a global iterator

Returns

iterator to the beginning of the local container's data.

inline auto **`begin`**`()` const

Returns a const_iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the beginning of the local container's data.

inline auto **`cbegin`**`()` const

Returns a const_iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the beginning of the local container's data.

inline auto **`end`**`()`

Returns an iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The iterator is a local iterator, not a global iterator

Returns

iterator to the end of the local container's data.

inline auto **end**() const

Returns a const_iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the end of the local container's data.

inline auto **cend**() const

Returns a const_iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the end of the local container's data.

inline void **for_all**(Function fn)

Iterates over all items in a container and executes a user-provided function object on each.

The user-provided function is expected to take a single key and value as separate arguments that make a (key, value) pair within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters

Function – Type of user-provided function

Parameters

fn – User-provided function

inline void **for_all**(Function &&fn) const

Const version of for_all that iterates over all items and passes them to the user function as const *.

The user-provided function is expected to take a single key and value as separate arguments that make a (key, value) pair within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters
`Function` – Type of user-provided function

Parameters
`fn` – User-provided function

inline void `gather`(STLContainer &gto, int rank) const

Gather all values in an STL container.

Requires STL container to have a `value_type` that is (key, value) pairs from the YGM container

Template Parameters
`STLContainer` – Type of STL container to gather to

Parameters

- `gto` – Container to store results in
- `rank` – Rank to tather results on. Use -1 to gather to all ranks

inline void `gather`(STLContainer &gto) const

Gather all values in an STL container on all ranks.

Equivalent to `gather(gto, -1)`

Template Parameters
`STLContainer` – Type of STL container to gather to

Parameters
`gto` – Container to store results in

inline std::vector<std::pair<*key_type*, *mapped_type*>> `gather_topk`(size_t k, Compare comp = Compare()) const

Gather the k “largest” key-value pairs according to provided comparison function.

Template Parameters
`Compare` – Type of comparison operator

Parameters

- `k` – Number of key-value pairs to gather
- `comp` – Comparison function for identifying elements to gather

Returns
vector of largest key-value pairs

inline void `collect`(YGMContainer &c) const

Collects all items in a new YGM container.

Template Parameters
`YGMContainer` – Container type

Parameters
`c` – Container to collect into

inline void **reduce_by_key**(MapType &map, ReductionOp reducer) const

Reduces all values in key-value pairs with matching keys.

Template Parameters

- `MapType` – Result YGM container type
- `ReductionOp` – Functor type

Parameters

- `map` – YGM container to hold result
- `reducer` – Functor for combining values

transform_proxy_key_value<*counting_set*<*Key*>, TransformFunction> **transform**(TransformFunction &&ffn)

Creates proxy that transforms key-value pairs in a container that are presented to user `for_all` calls.

The underlying items within the container are not modified.

Template Parameters

`TransformFunction` – functor type

Parameters

`ffn` – Function to transform items in container

inline auto **keys**()

Access to container presenting only keys.

Returns

Transform object that returns only keys to user

inline auto **values**()

Access to container presenting only values.

Returns

Transform object that returns only values to user

inline flatten_proxy_key_value<*counting_set*<*Key*>> **flatten**()

Flattens STL containers of values to allow a function to be called on inner items individually.

Underlying container is not modified.

filter_proxy_key_value<*counting_set*<*Key*>, FilterFunction> **filter**(FilterFunction &&ffn)

Filters items in a container so only allow `for_all` to execute on those that satisfy a given predicate function.

Filtered items are not removed from underlying container.

Template Parameters

`FilterFunction` – Functor type

Parameters

`ffn` – Function used to filter items in container.

Public Members

const *size_type* **count_cache_size** = 1024 * 1024

detail::hash_partitioner<detail::hash<*key_type*>> **partitioner**

Friends

friend class detail::base_misc< counting_set< Key >, std::tuple< Key, size_t > >

3.5.4 disjoint_set

template<typename **Item**, typename **Partitioner** = detail::old_hash_partitioner<*Item*>>

class **disjoint_set**

Public Types

using **self_type** = *disjoint_set*<*Item*, *Partitioner*>

using **value_type** = *Item*

using **size_type** = size_t

using **ygm_for_all_types** = std::tuple<*Item*, *Item*>

using **container_type** = ygm::container::disjoint_set_tag

using **impl_type** = detail::disjoint_set_impl<*Item*, *Partitioner*>

Public Functions

disjoint_set() = delete

inline **disjoint_set**(ygm::*comm* &comm, const size_t cache_size = 8192)

template<typename **Visitor**, typename ...**VisitorArgs**>
inline void **async_visit**(const *value_type* &item, *Visitor* visitor, const *VisitorArgs*&... args)

inline void **async_union**(const *value_type* &a, const *value_type* &b)

template<typename **Function**, typename ...**FunctionArgs**>
inline void **async_union_and_execute**(const *value_type* &a, const *value_type* &b, *Function* fn, const *FunctionArgs*&... args)

inline void **all_compress**()

template<typename **Function**>
inline void **for_all**(*Function* fn)

inline std::map<*value_type*, *value_type*> **all_find**(const std::vector<*value_type*> &items)

inline void **clear**()

inline *size_type* **size**()

inline *size_type* **num_sets**()

inline ygm::ygm_ptr<*impl_type*> **get_ygm_ptr**() const

inline ygm::*comm* &**comm**()

3.5.5 map

template<typename **Key**, typename **Value**>

class **map** : public ygm::container::detail::base_async_insert_key_value<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>, public ygm::container::detail::base_async_insert_or_assign<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>, public ygm::container::detail::base_misc<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>, public ygm::container::detail::base_count<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>, public ygm::container::detail::base_async_reduce<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>, public ygm::container::detail::base_async_erase_key<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>, public ygm::container::detail::base_async_erase_key_value<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>, public ygm::container::detail::base_batch_erase_key_value<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>, public ygm::container::detail::base_async_visit<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>, public ygm::container::detail::base_iterators<*map*<*Key*, *Value*>>, public ygm::container::detail::base_iteration_key_value<*map*<*Key*, *Value*>, std::tuple<*Key*, *Value*>>

Public Types

using **self_type** = *map*<*Key*, *Value*>

using **mapped_type** = *Value*

using **ptr_type** = typename ygm::ygm_ptr<*self_type*>

using **key_type** = *Key*

using **size_type** = size_t

using **for_all_args** = std::tuple<*Key*, *Value*>

using **container_type** = ygm::container::map_tag using `iterator` = typename local_container_type::iterator

using `const_iterator` = typename local_container_type::const_iterator

Public Functions

`map`() = delete

inline `map`(ygm::*comm* &comm)

Map constructor.

Parameters

`comm` – Communicator to use for communication

inline `map`(ygm::*comm* &comm, const *mapped_type* &default_value)

Map constructor taking default value.

Parameters

- `comm` – Communicator to use for communication
- `default_value` – Value to initialize all stored items with

inline `map`(ygm::*comm* &comm, std::initializer_list<std::pair<*Key*, *Value*>> l)

Map constructor from std::initializer_list of key-value pairs.

Initializer list is assumed to be replicated on all ranks.

Parameters

- `comm` – Communicator to use for communication
- `l` – Initializer list of key-value pairs to put in map

template<typename `STLContainer`>
inline `map`(ygm::*comm* &comm, const *STLContainer* &cont)
requires detail

Construct map from existing STL container.

Template Parameters

`T` – Existing container type

Parameters

- `comm` – Communicator to use for communication
- `cont` – STL container containing key-value pairs to put in map

template<typename `YGMContainer`>
inline `map`(ygm::*comm* &comm, const *YGMContainer* &yc)
requires detail

Construct map from existing YGM container of key-value pairs.

Requires input container `for_all_args` to be a single item that is itself a key-value pair.

Template Parameters

`T` – Existing container type

Parameters

- **comm** – Communicator to use for communication
- **yc** – YGM container of key-value pairs to put in map.

inline **~map**()

inline **map**(const *self_type* &other)

inline **map**(*self_type* &&other) noexcept

inline *map* &**operator=**(const *self_type* &other)

inline *map* &**operator=**(*self_type* &&other)

inline *iterator* **local_begin**()

Access to begin iterator of locally-held items.

Does not call `barrier()`.

Returns

Local iterator to beginning of items held by process.

inline *const_iterator* **local_begin**() const

Access to begin const_iterator of locally-held items for const map.

Does not call `barrier()`.

Returns

Local const iterator to beginning of items held by process.

inline *const_iterator* **local_cbegin**() const

Access to begin const_iterator of locally-held items for const map.

Does not call `barrier()`.

Returns

Local const iterator to beginning of items held by process.

inline *iterator* **local_end**()

Access to end iterator of locally-held items.

Does not call `barrier()`.

Returns

Local iterator to ending of items held by process.

inline *const_iterator* **local_end**() const

Access to end const_iterator of locally-held items for const map.

Does not call `barrier()`.

Returns

Local const iterator to ending of items held by process.

inline *const_iterator* **`local_cend`**`()` const

Access to end const_iterator of locally-held items for const map.

Does not call `barrier()`.

Returns

Local const iterator to ending of items held by process.

inline void **`local_insert`**`(`const *key_type* &key`)`

Insert a key and default value into local storage.

Parameters

`key` – Local index to store default value at

inline void **`local_erase`**`(`const *key_type* &key`)`

Erase local entry for given key.

Parameters

`key` – Key to erase from local storage

inline void **`local_erase`**`(`const *key_type* &key, const *key_type* &value`)`

Erase local entry for given key and value.

Does not erase the entry if key is found with a different value

Parameters

- **`key`** – Key to erase from local storage
- **`value`** – Value to erase if associated to key

inline void **`local_insert`**`(`const *key_type* &key, const *mapped_type* &value`)`

Insert a key and value into local storage.

Parameters

- **`key`** – Local index to store value at
- **`value`** – Value to store

inline void **`local_insert_or_assign`**`(`const *key_type* &key, const *mapped_type* &value`)`

Insert a key and value into local storage or assign value to key if key is already present.

Parameters

- **`key`** – Local index to store value at
- **`value`** – Value to store

inline void **`local_clear`**`()`

Clear local storage.

template<typename **`ReductionOp`**>
inline void **`local_reduce`**`(`const *key_type* &key, const *mapped_type* &value, *ReductionOp* reducer`)`

Update a locally stored element by performing a binary operation between it and a provided value.

Template Parameters

`ReductionOp` – functor type

Parameters

- **key** – Key to perform binary operation at.
- **value** – Value to combine with the currently-held value
- **reducer** – Binary operation to perform

inline size_t **local_size**() const

Get the number of elements stored on the local process.

Returns

Local size of map

inline *mapped_type* &**local_at**(const *key_type* &key)

Retrieve value for given key.

Throws an exception if key is not found in local storage

Parameters

key – Key to look up value for

inline const *mapped_type* &**local_at**(const *key_type* &key) const

Retrieve const reference to value for given key.

Throws an exception if key is not found in local storage

Parameters

key – Key to look up value for

template<typename **Function**, typename ...**VisitorArgs**>
inline void **local_visit**(const *key_type* &key, *Function* &&fn, const *VisitorArgs*&... args)

Visit a key-value pair stored locally.

Template Parameters

- **Function** – functor type
- **VisitorArgs...** – Variadic argument types

Parameters

- **key** – Key to visit
- **fn** – User-provided function to execute at item
- **args...** – Arguments to pass to user functor

template<typename **Function**, typename ...**VisitorArgs**>
inline void **local_visit_if_contains**(const *key_type* &key, *Function* &&fn, const *VisitorArgs*&... args)

Visit a key-value pair stored locally if the key is found.

Does not create an entry if `key` is not already found in local map

Template Parameters

- **Function** – functor type
- **VisitorArgs...** – Variadic argument types

Parameters

- **key** – Key to visit

- **fn** – User-provided function to execute at item
- **args...** – Arguments to pass to user functor

template<typename **Function**, typename ...**VisitorArgs**>
inline void **local_visit_if_contains**(const *key_type* &key, *Function* &&fn, const *VisitorArgs*&... args) const

`local_visit_if_contains` for `const` containers

Does not create an entry if `key` is not already found in local map. `fn` is given a `const` key and value for execution.

Template Parameters

- **Function** – functor type
- **VisitorArgs...** – Variadic argument types

Parameters

- **key** – Key to visit
- **fn** – User-provided function to execute at item
- **args...** – Arguments to pass to user functor

template<typename **STLKeyContainer**>
inline std::map<*key_type*, *mapped_type*> **gather_keys**(const *STLKeyContainer* &keys)

Collective operation to look up item counts from each rank.

Parameters
keys – Keys local rank wants to collect counts for

Returns
`std::map` of provided keys and their counts

inline std::vector<*mapped_type*> **local_get**(const *key_type* &key) const

Retrieve all values associated to a given key.

Currently, *`ygm::container::map`* is not a multi-map, so there can be at most one value associated to each key.

Parameters
key – Key to retrieve values for

Returns
Vector of values associated to key

template<typename **Function**>
inline void **local_for_all**(*Function* &&fn)

Execute a functor on every locally-held key and value.

Template Parameters
Function – functor type

Parameters
fn – Functor to execute on keys and values

template<typename **Function**>

inline void **local_for_all**(*Function* &&fn) const

local_for_all for const containers

const references to key and value are provided to fn

Template Parameters
Function – functor type

Parameters
fn – Functor to execute on keys and values

inline size_t **local_count**(const *key_type* &key) const

Count the number of times a given key is found locally.

Returns
Number of times key is found locally

inline void **local_swap**(*self_type* &other)

Swap the local contents of a map.

Parameters
other – The map to swap local contents with

inline void **async_insert**(const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type &key, const typename std::tuple_element<1, std::tuple<*Key*, *Value*>>::type &value)
requires DoubleItemTuple<std::tuple<*Key*, *Value*>>

Asynchronously insert a key-value pair into a container.

The container's local_insert() function is free to determine the behavior when key is already in the container

```
ygm::container::map<int, std::string> my_map(world);
my_map.async_insert(1, "one");
```

Parameters

- **key** – Key to insert
- **value** – Value to associate to key

inline void **async_insert**(const std::pair<const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type, typename std::tuple_element<1, std::tuple<*Key*, *Value*>>::type> &kvp)
requires DoubleItemTuple<std::tuple<*Key*, *Value*>>

Asynchronously insert a key-value pair into a container.

Equivalent to async_insert(kvp.first, kvp.second)

Parameters
kvp – Key-value pair to insert

inline void **async_insert_or_assign**(const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type &key, const typename std::tuple_element<1, std::tuple<*Key*, *Value*>>::type &value)

requires DoubleItemTuple<std::tuple<*Key*, *Value*>>

Asynchronously insert `(key, value)` pair into container if it does not already exist or assign `value` to `key` if `key` already exists in the container.

Behavior is meant to mirror `std::map::insert_or_assign`

Parameters

- **`key`** – Key to attempt insertion of
- **`value`** – Value to associate with key

inline void **`async_insert_or_assign`**(const std::pair<typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type, typename std::tuple_element<1, std::tuple<*Key*, *Value*>>::type> &kvp)
requires DoubleItemTuple<std::tuple<*Key*, *Value*>>

Asynchronously insert `(key, value)` pair into container if it does not already exist or assign `value` to `key` if `key` already exists in the container.

Equivalent to `async_insert_or_assign(kvp.first, kvp.second)`

Parameters

`kvp` – Key-value pair to attempt to insert

inline size_t **`size`**() const

Gets number of elements in a YGM container.

Returns

Container size

inline void **`clear`**()

Clears the contents of a YGM container.

inline void **`swap`**(*map*<*Key*, *Value*> &other)

Swaps the contents of a YGM container.

Parameters

`other` – Container to swap with

inline ygm::*comm* &**`comm`**() const

Access to underlying YGM communicator.

Returns

YGM communicator used for communication by container

inline ygm::ygm_ptr<*map*<*Key*, *Value*>> **`get_ygm_ptr`**()

Access to the ygm_ptr used by the container.

Returns

`ygm_ptr` used by the container when identifying itself in `async` calls on the *`ygm::comm`*

inline const ygm::ygm_ptr<*map*<*Key*, *Value*>> **`get_ygm_ptr`**() const

Const access to the ygm ptr used by the container.

Returns

`ygm_ptr` to const version of container

inline size_t **count**(const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type &value) const

Counts all occurrences of a value within a container.

Parameters

value – Value to search for within container (key in the case of containers with keys)

Returns

Count of times `value` is seen in container

inline void **async_reduce**(const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type &key, const typename std::tuple_element<1, std::tuple<*Key*, *Value*>>::type &value, ReductionOp reducer)

Combines existing `mapped_type` item with `value` using a user-provided binary operation if `key` is found in container. Inserts default `mapped_type` prior to reduction if `key` does not already exist in container.

```
ygm::container::map<std::string, int> my_map(world);
my_map.async_insert("one", 1);
if (world.rank0()) {
   my_map.async_reduce("one", 2, std::plus<int>());
   my_map.async_reduce("two", 2, std::plus<int>());
}
world.barrier()
```

will result in `my_map` containing the pairs `("one", 3)` and `("two", 2)`.

Template Parameters

ReductionOp – Type of function provided by usert to perform reduction

Parameters

- **key** – Key to search for within container
- **value** – Provided value to combine with existing value in container

inline void **async_erase**(const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type &key)
requires AtLeastOneItemTuple<std::tuple<*Key*, *Value*>>

Asynchronously erases a key from a container.

Parameters

key – Key to erase (key, value) pair of in containers with keys and values or value to erase in containers without keys

inline void **async_erase**(const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type &key, const typename std::tuple_element<1, std::tuple<*Key*, *Value*>>::type &value)
requires DoubleItemTuple<std::tuple<*Key*, *Value*>>

Asynchronously erases key and value from a container.

Does nothing if (key, value) pair is not found.

Parameters

- **key** – Key to find in container
- **value** – Value to find associated to key

inline void **async_visit**(const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type &key, Visitor &&visitor, const VisitorArgs&... args) requires DoubleItemTuple<std::tuple<*Key*, *Value*>>

Asynchronously visit key within a container and execute a user-provided function.

```
ygm::container::map<std::string, int> my_map(world);
my_map.async_insert("one", 1);
world.barrier();
my_map.async_visit("one", [](const auto &key, auto &val, int &to_add) {
   val += to_add;
   }, world.size())
world.barrier();
```

will result in a value of `world.size() * world.size() + 1` associated to the key `"one"`.

Template Parameters

- **`Visitor`** – Type of user-provided function
- **`VisitorArgs...`** – Variadic argument types to give to user function

Parameters

- **`key`** – Key to visit in container
- **`visitor`** – User-provided function to execute at key
- **`args...`** – Variadic arguments to pass to user-provided function

inline void **`async_visit_if_contains`**(const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type &key, Visitor visitor, const VisitorArgs&... args)
requires DoubleItemTuple<std::tuple<*Key*, *Value*>>

Asynchronously visit key within a container and execute a user-provided function only if the key already exists.

This function differs from `async_visit` in that it will not default construct a value within the container prior to visiting a key that does not already exist.

Template Parameters

- **`Visitor`** – Type of user-provided function
- **`VisitorArgs...`** – Variadic argument types to give to user function

Parameters

- **`key`** – Key to visit in container
- **`visitor`** – User-provided function to execute at key
- **`args...`** – Variadic arguments to pass to user-provided function

inline void **`async_visit_if_contains`**(const typename std::tuple_element<0, std::tuple<*Key*, *Value*>>::type &key, Visitor visitor, const VisitorArgs&... args) const
requires DoubleItemTuple<std::tuple<*Key*, *Value*>>

Version of `async_visit_if_contains` that works on `const` objects and provides `const` arguments to the user-provided lambda.

inline auto **begin**()

Returns an iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The iterator is a local iterator, not a global iterator

Returns

iterator to the beginning of the local container's data.

inline auto **begin**() const

Returns a const_iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the beginning of the local container's data.

inline auto **cbegin**() const

Returns a const_iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the beginning of the local container's data.

inline auto **end**()

Returns an iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The iterator is a local iterator, not a global iterator

Returns

iterator to the end of the local container's data.

inline auto **end**() const

Returns a const_iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the end of the local container's data.

inline auto **cend**() const

Returns a const_iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the end of the local container's data.

inline void **for_all**(Function fn)

Iterates over all items in a container and executes a user-provided function object on each.

The user-provided function is expected to take a single key and value as separate arguments that make a (key, value) pair within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters

Function – Type of user-provided function

Parameters

fn – User-provided function

inline void **for_all**(Function &&fn) const

Const version of for_all that iterates over all items and passes them to the user function as const *.

The user-provided function is expected to take a single key and value as separate arguments that make a (key, value) pair within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters

Function – Type of user-provided function

Parameters
`fn` – User-provided function

inline void **`gather`**(STLContainer &gto, int rank) const

Gather all values in an STL container.

Requires STL container to have a `value_type` that is (key, value) pairs from the YGM container

Template Parameters
`STLContainer` – Type of STL container to gather to

Parameters

- **`gto`** – Container to store results in
- **`rank`** – Rank to tather results on. Use `-1` to gather to all ranks

inline void **`gather`**(STLContainer &gto) const

Gather all values in an STL container on all ranks.

Equivalent to `gather(gto, -1)`

Template Parameters
`STLContainer` – Type of STL container to gather to

Parameters
`gto` – Container to store results in

inline std::vector<std::pair<*key_type*, *mapped_type*>> **`gather_topk`**(size_t k, Compare comp = Compare()) const

Gather the k "largest" key-value pairs according to provided comparison function.

Template Parameters
`Compare` – Type of comparison operator

Parameters

- **`k`** – Number of key-value pairs to gather
- **`comp`** – Comparison function for identifying elements to gather

Returns
vector of largest key-value pairs

inline void **`collect`**(YGMContainer &c) const

Collects all items in a new YGM container.

Template Parameters
`YGMContainer` – Container type

Parameters
`c` – Container to collect into

inline void **`reduce_by_key`**(MapType &map, ReductionOp reducer) const

Reduces all values in key-value pairs with matching keys.

Template Parameters

- **`MapType`** – Result YGM container type
- **`ReductionOp`** – Functor type

Parameters

- `map` – YGM container to hold result
- `reducer` – Functor for combining values

transform_proxy_key_value<*map*<*Key*, *Value*>, TransformFunction> **`transform`**(TransformFunction &&ffn)

Creates proxy that transforms key-value pairs in a container that are presented to user `for_all` calls.

The underlying items within the container are not modified.

Template Parameters

`TransformFunction` – functor type

Parameters

`ffn` – Function to transform items in container

inline auto **`keys`**()

Access to container presenting only keys.

Returns

Transform object that returns only keys to user

inline auto **`values`**()

Access to container presenting only values.

Returns

Transform object that returns only values to user

inline flatten_proxy_key_value<*map*<*Key*, *Value*>> **`flatten`**()

Flattens STL containers of values to allow a function to be called on inner items individually.

Underlying container is not modified.

filter_proxy_key_value<*map*<*Key*, *Value*>, FilterFunction> **`filter`**(FilterFunction &&ffn)

Filters items in a container so only allow `for_all` to execute on those that satisfy a given predicate function.

Filtered items are not removed from underlying container.

Template Parameters

`FilterFunction` – Functor type

Parameters

`ffn` – Function used to filter items in container.

Public Members

detail::hash_partitioner<detail::hash<*key_type*>> **`partitioner`**

Friends

friend class detail::base_misc< map< Key, Value >, std::tuple< Key, Value > >

3.5.6 set

template<typename **Value**>

class **set** : public ygm::container::detail::base_async_insert_value<*set*<*Value*>, std::tuple<*Value*>>, public ygm::container::detail::base_async_erase_key<*set*<*Value*>, std::tuple<*Value*>>, public ygm::container::detail::base_batch_erase_key<*set*<*Value*>, std::tuple<*Value*>>, public ygm::container::detail::base_async_contains<*set*<*Value*>, std::tuple<*Value*>>, public ygm::container::detail::base_async_insert_contains<*set*<*Value*>, std::tuple<*Value*>>, public ygm::container::detail::base_count<*set*<*Value*>, std::tuple<*Value*>>, public ygm::container::detail::base_misc<*set*<*Value*>, std::tuple<*Value*>>, public ygm::container::detail::base_iterators<*set*<*Value*>>, public ygm::container::detail::base_iteration_value<*set*<*Value*>, std::tuple<*Value*>>

Public Types

using **self_type** = *set*<*Value*>

using **value_type** = *Value*

using **size_type** = size_t

using **for_all_args** = std::tuple<*Value*>

using **container_type** = ygm::container::set_tag

using **iterator** = typename local_container_type::iterator

using **const_iterator** = typename local_container_type::const_iterator

using **key_type** = typename std::tuple_element_t<0, std::tuple<*Value*>>

Public Functions

inline **set**(ygm::*comm* &comm)

Set constructor.

Parameters

comm – Communicator to use for communication inline **set**(ygm::*comm* &comm, std::initializer_list<*Value*> l)

Set constructor from std::initializer_list of sets.

Initializer list is assumed to be replicated on all ranks.

Parameters

- **comm** – Communicator to use for communication
- **l** – Initializer list of values to put in set

template<typename **STLContainer**>
inline **set**(ygm::*comm* &comm, const *STLContainer* &cont)
requires detail

Construct set from existing STL container.

Template Parameters

T – Existing container type

Parameters

- **comm** – Communicator to use for communication
- **cont** – STL container containing values to put in set

template<typename **YGMContainer**>
inline **set**(ygm::*comm* &comm, const *YGMContainer* &yc)
requires detail

Construct set from existing YGM container of values.

Template Parameters

T – Existing container type

Parameters

- **comm** – Communicator to use for communication
- **yc** – YGM container of values to put in set.

inline **~set**()

set() = delete

inline **set**(const *self_type* &other)

inline **set**(*self_type* &&other) noexcept

inline *set* &**operator=**(const *self_type* &other)

inline *set* &**operator=**(*self_type* &&other) noexcept

inline *iterator* **local_begin**()

Access to begin iterator of locally-held items.

Does not call `barrier()`.

Returns

Local iterator to beginning of items held by process.

inline *const_iterator* **`local_begin`**`()` const

Access to begin const_iterator of locally-held items for const set.

Does not call `barrier()`.

Returns

Local const iterator to beginning of items held by process.

inline *const_iterator* **`local_cbegin`**`()` const

Access to begin const_iterator of locally-held items for const set.

Does not call `barrier()`.

Returns

Local const iterator to beginning of items held by process.

inline *iterator* **`local_end`**`()`

Access to end iterator of locally-held items.

Does not call `barrier()`.

Returns

Local iterator to ending of items held by process.

inline *const_iterator* **`local_end`**`()` const

Access to end const_iterator of locally-held items for const set.

Does not call `barrier()`.

Returns

Local const iterator to ending of items held by process.

inline *const_iterator* **`local_cend`**`()` const

Access to end const_iterator of locally-held items for const set.

Does not call `barrier()`.

Returns

Local const iterator to ending of items held by process.

inline void **`local_insert`**`(`const *value_type* &val`)`

Insert a value into local storage.

Parameters

`val` – Value to store

inline void **`local_erase`**`(`const *value_type* &val`)`

Erase value from local storage.

Parameters

`val` – Value to erase from local storage

inline void **`local_clear`**`()`

Clear local storage.

inline size_t **`local_count`**`(`const *value_type* &val`)` const

Count the number of times a value is found locally.

Returns

Number of local occurrences of `val`

inline size_t **`local_size`**`()` const

Get the number of elements stored on the local process.

Returns

Local size of set

template<typename **`Function`**>
inline void **`local_for_all`**`(`*Function* &&fn`)`

Execute a functor on every locally-held value.

Template Parameters

`Function` – functor type

Parameters

`fn` – Functor to execute on values

template<typename **`Function`**>
inline void **`local_for_all`**`(`*Function* &&fn`)` const

`local_for_all` for `const` containers

`const` references to values are provided to `fn`

Template Parameters

`Function` – functor type

Parameters

`fn` – Functor to execute on values

inline void **`serialize`**`(`const std::string &fname`)`

Serialize a set to a collection of files to be read back in later.

Parameters

`fname` – Filename prefix to create filename used by every rank from

inline void **`deserialize`**`(`const std::string &fname`)`

Deserialize a set from files.

Currently requires the number of ranks deserializing a bag to be the same as was used for serialization.

Parameters

`fname` – Filename prefix to create filename used by every rank from

inline void **`local_swap`**`(`*self_type* &other`)`

Swap elements held locally between sets.

Parameters

`other` – Set to swap elements with

inline void **`async_insert`**`(`const typename std::tuple_element<0, std::tuple<*Value*>>::type &value`)`
requires SingleItemTuple<std::tuple<*Value*>>

Asynchronously inserts a value in a container.

```
ygm::container::bag<int> my_bag(world);
my_bag.async_insert(world.rank());
```

Parameters

value – Value to insert into container

inline void **async_erase**(const typename std::tuple_element<0, std::tuple<*Value*>>::type &key) requires AtLeastOneItemTuple<std::tuple<*Value*>>

Asynchronously erases a key from a container.

Parameters

key – Key to erase (key, value) pair of in containers with keys and values or value to erase in containers without keys

inline void **async_contains**(const typename std::tuple_element<0, std::tuple<*Value*>>::type &value, Function &&fn, const FuncArgs&... args)

Asynchronously execute a function with knowledge of if the container contains the given value.

The user-provided function is provided with (1) an optional pointer to the container, (2) a boolean indicating whether the desired value was found, (3) the value searched for, and (4) any additional arguments passed to async_contains by the user

Template Parameters

- **Function** – Type of user function
- **FuncArgs...** – Variadic types of user-provided arguments to function

Parameters

- **value** – Value to check for existence of in container
- **fn** – User-provided function to execute
- **args...** – Variadic arguments to user-provided function

inline void **async_insert_contains**(const typename std::tuple_element<0, std::tuple<*Value*>>::type &value, Function &&fn, const FuncArgs&... args)

Asynchronously insert into a container if value is not already present and execute a user-provided function that is told whether the value was already present.

Insertion only occurs if value is not already present. Containers with keys and values will not have values reset to the value's default.

```
ygm::container::map<int, int> my_map(world);
my_bag.async_insert_contains(10, [](bool contains, auto &value) {
   if (contains) {
     wcout() << "my_map already contained " << value << std::endl;
   } else {
     wcout() << "my_map did not already contain " << value << " but now it
does" << std::endl;
   }
 });
```

Parameters

- `value` – Value to attempt to insert
- `fn` – Function to execute after attempted insertion
- `args...` – Variadic arguments to pass to fn

inline size_t **count**(const typename std::tuple_element<0, std::tuple<*Value*>>::type &value) const

Counts all occurrences of a value within a container.

Parameters

`value` – Value to search for within container (key in the case of containers with keys)

Returns

Count of times `value` is seen in container

inline size_t **size**() const

Gets number of elements in a YGM container.

Returns

Container size

inline void **clear**()

Clears the contents of a YGM container.

inline void **swap**(*set*<*Value*> &other)

Swaps the contents of a YGM container.

Parameters

`other` – Container to swap with

inline ygm::*comm* &**comm**() const

Access to underlying YGM communicator.

Returns

YGM communicator used for communication by container

inline ygm::ygm_ptr<*set*<*Value*>> **get_ygm_ptr**()

Access to the ygm_ptr used by the container.

Returns

`ygm_ptr` used by the container when identifying itself in `async` calls on the *`ygm::comm`*

inline const ygm::ygm_ptr<*set*<*Value*>> **get_ygm_ptr**() const

Const access to the ygm ptr used by the container.

Returns

`ygm_ptr` to const version of container

inline auto **begin**()

Returns an iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The iterator is a local iterator, not a global iterator

Returns

iterator to the beginning of the local container's data.

inline auto **begin**() const

Returns a const_iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the beginning of the local container's data.

inline auto **cbegin**() const

Returns a const_iterator to the beginning of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the beginning of the local container's data.

inline auto **end**()

Returns an iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

> **Warning**
>
> The iterator is a local iterator, not a global iterator

Returns

iterator to the end of the local container's data.

inline auto **end**() const

Returns a const_iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the end of the local container's data.

inline auto `cend()` const

Returns a const_iterator to the end of the local container's data.

This function is primarily a convienience function for range based for loops

Warning

The const_iterator is a local iterator, not a global iterator

Returns

auto const_iterator to the end of the local container's data.

inline void `for_all`(Function &&fn)

Iterates over all items in a container and executes a user-provided function object on each.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters

`Function` – Type of user-provided function

Parameters

`fn` – User-provided function

inline void `for_all`(Function &&fn) const

Const version of for_all that iterates over all items and passes them to the user function as const *.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters

`Function` – Type of user-provided function

Parameters

`fn` – User-provided function

inline void `gather`(STLContainer &gto, int rank) const

Gather all values in an STL container.

Template Parameters

`STLContainer` – Type of STL container to gather to

Parameters

- `gto` – Container to store results in

- **rank** – Rank to tather results on. Use -1 to gather to all ranks

inline void **gather**(STLContainer &gto) const

Gather all values in an STL container on all ranks.

Equivalent to `gather(gto, -1)`

Template Parameters
STLContainer – Type of STL container to gather to

Parameters
gto – Container to store results in

inline std::vector<*value_type*> **gather_topk**(size_t k, Compare comp = std::greater<*value_type*>()) const requires SingleItemTuple<*for_all_args*>

Gather the k "largest" values according to provided comparison function.

Template Parameters
Compare – Type of comparison operator

Parameters

- **k** – Number of values to gather
- **comp** – Comparison function for identifying elements to gather

Returns
vector of largest values

inline *value_type* **reduce**(MergeFunction merge) const

Perform a reduction over all items in container.

`reduce` only makes sense to use with commutative and associative functors defining merges. Otherwise, ranks will not receive the same result.

Template Parameters
MergeFunction – Merge functor type

Parameters
merge – Functor to combine pairs of items

Returns
Value from all reductions

inline void **collect**(YGMContainer &c) const

Collects all items in a new YGM container.

Template Parameters
YGMContainer – Container type

Parameters
c – Container to collect into

inline void **reduce_by_key**(MapType &map, ReductionOp reducer) const

Reduces all values in key-value pairs with matching keys.

Template Parameters

- **MapType** – Result YGM container type
- **ReductionOp** – Functor type

Parameters

- **map** – YGM container to hold result
- **reducer** – Functor for combining values

transform_proxy_value<*set*<*Value*>, TransformFunction> **transform**(TransformFunction &&ffn)

Creates proxy that transforms items in container that are presented to user `for_all` calls.

The underlying items within the container are not modified.

```
ygm::container::bag<int> my_bag(world);
my_bag.async_insert(2);
my_bag.barrier();

my_bag.transform([](auto &val) { return 2*val; }).for_all([](const auto
&transformed_val) { YGM_ASSERT_RELEASE(val == 4);
});

my_bag.for_all([](const auto &val) { YGM_ASSERT_RELEASE(val == 2); });
```

will complete successfully.

Template Parameters

TransformFunction – functor type

Parameters

ffn – Function to transform items in container

inline flatten_proxy_value<*set*<*Value*>> **flatten**()

Flattens STL containers of values to allow a function to be called on inner items individually.

Underlying container is not modified.

```
ygm::container::bag<std::vector<int>> my_bag(world, {{1, 2, 3}});

my_bag.flatten().for_all([](const int &nested_val) {
std::cout << "Nested value: " << nested_val << std::cout;
});
```

will print

```
Nested value: 1
Nested value: 2
Nested value: 3
```

filter_proxy_value<*set*<*Value*>, FilterFunction> **filter**(FilterFunction &&ffn)

Filters items in a container so only allow `for_all` to execute on those that satisfy a given predicate function.

Filtered items are not removed from underlying container.

```
ygm::container::bag<int> my_bag(world, {1, 2, 3, 4});
my_bag.filter([](const auto &val) { return (val % 2) == 0;
```

(continues on next page)

(continued from previous page)

```
}).for_all([](const auto &filtered_val) { YGM_ASSERT_RELEASE((filtered_val %
2) == 0);
});
```

Template Parameters
`FilterFunction` – Functor type

Parameters
`ffn` – Function used to filter items in container.

Public Members

detail::hash_partitioner<detail::hash<*value_type*>> **partitioner**

Friends

friend class detail::base_misc< set< Value >, std::tuple< Value > >

CHAPTER

FOUR

YGM::IO MODULE REFERENCE

The ygm::io module provides parallel I/O functionality for use with YGM's communicator. This allows for simple parallel reading of large (collections of) files where each line can be read independently of all others and writing of output to collections of files.

4.1 Reading Input

The reading functionality of YGM is built around the ygm::io::line_parser object. The for_all method of the line parser takes a lambda that is executed on every line of text within the files. As an example, the following code will read through file1.txt and file2.txt and count the lines that contain more than 10 characters:

```
ygm::io::line_parser my_line_parser(world, {"file1.txt", "file2.txt"});

int long_line_count{0};
my_line_parser.for_all([&long_line_count](const std::string &line) {
  if (line.size() > 10) {
    ++long_line_count;
  });

long_line_count = ygm::sum(long_line_count, world);
```

The line parser assigns contiguous chunks of the files being read to all ranks in the communicator (with a minimum size to avoid partitioning files into unrealistically small pieces). This splitting is done based on the number of bytes within files, with starting positions adjusted to the nearest newline. For this reason, it must be possible to process each line of the input files independently of all others, and there is not support for more complicated record parsing.

YGM has parsers (often built on top of the ygm::io::line_parser) for when data is provided in specific formats. These function in much the same way as the line_parser, but do not require as much manual parsing of individual lines.

4.1.1 CSV Parser

The ygm::io::csv_parser takes each line of input and parses it into a csv_line object before it is provided to the user's lambda in a for_all call. This parsing converts all comma-separated values within a line into positional arguments that can be accessed from the csv_line and converted into various types. As an example, the following code reads all values within a line, checks to make sure they are usable as doubles, converts them to doubles, adds them up, and prints the result. This example also sums up the final entry in each column:

```
ygm::io::csv_parser my_csv_parser(world, {"file1.csv", "file2.csv"});

double final_sum{0.0};
my_csv_parser.for_all([&final_sum](const auto &line_csv) {
   double line_total;
   for (auto &entry : line_csv) {
     assert(entry.is_double());
     line_total += entry.as_double();
   }
   final_sum += line_csv[line_csv.size()-1];
   std::cout << "Line total: " << line_total << std::endl;
 });

 final_sum = ygm::sum(final_sum, world);
```

The entries within a parsed CSV line are stored as `csv_field` types. The following shows all of the available methods for checking the types of fields and converting them to primitive types:

class **`csv_field`**

Public Functions

inline **`csv_field`**(const std::string &f)

inline bool **`is_integer`**() const

inline int64_t **`as_integer`**() const

inline bool **`is_unsigned_integer`**() const

inline uint64_t **`as_unsigned_integer`**() const

inline bool **`is_double`**() const

inline double **`as_double`**() const

inline const std::string &**`as_string`**() const

CSVs with Headers

Many CSV files contain header lines that provide meaningful names to the columns of a file. For cases like these, the `ygm::io::csv_parser` has a `read_headers` method that reads the first line of the CSV files as a collection of column headers and provides named access to the columns in subsequent `for_all` calls. For example, we can sum values in `important_column1` and `important_column2` in CSV files containing columns named `important_column1`, `other_column`, and `important_column2` as follows:

```
 ygm::io::csv_parser my_csv_parser(world, {"file1.csv", "file2.csv", "file3.csv"});
 my_csv_parser.read_headers();

 double important_sum{0.0};
 my_csv_parser.for_all([&important_sum](const auto &line_csv) {
  important_sum += line_csv["important_column1"].as_double();
```

(continues on next page)

(continued from previous page)

```
  important_sum += line_csv["important_column2"].as_double();
});
```

When reading CSV files with headers, it is important to remember that

- all CSV files provided must contain headers that are identical
- if a CSV file with headers is read without calling `read_headers()` the header line will be treated as a normal line with data

4.1.2 NDJSON Parser

The `ygm::io::ndjson_parser` handles lines of input that are provided as newline-delimited JSON (NDJSON), a.k.a. JSON lines data. JSON support is provided by Boost JSON and requires some knowledge of the associated syntax. To sum the `number` field in all JSON records as integers, we can do the following:

```
ygm::io::ndjson_parser my_json_parser(world, {"file1.ndjson"});

int64_t total{0};
my_ndjson_parser.for_all([&total](const auto &json_line) {
 if (json_line["number"].is_int64()) {
   json_line["number"].as_int64();
 }
});

total = ygm::sum(total, world);
```

4.1.3 Parquet Parser

YGM provides Parquet parsing through the use of Apache Arrow in its `ygm::io::parquet_parser`. A row of data is provided to a `for_all` operation as a `vector` of data entries provided as a `variant`. Optionally, a `vector` of column names can be provided as to specify the set of columns needed by the lambda being executed on the rows. If no columns names are provided, the default behavior is to provide all columns to the lambda. To print the “string_column” and “float_column” columns of a Parquet dataset, use:

```
ygm::io::parquet_parser my_parquet_parser(world, {"file.parquet"});

my_parquet_parser.for_all({"string_column", "float_column"}(const auto &row_values) {
 std::cout << std::get<std::string>(row_values[0]) << "\t" << std::get<float>(row_
↪values[1]) << std::endl;
});
```

4.2 Writing Output

When writing large amounts of output, there are two main ways of doing so in YGM. The simplest and most frequently encountered is where output is written in a manner that does not require organization. In these situations, it is easiest to have every rank open a separate file (using `std::ofstream`) that is distinct from files on all other ranks for writing its own local data.

The second supported way of writing files is when output generated anywhere on the system has a natural filename that it must be found in. In this case, independent ranks cannot open all files and write to them safely. For such use cases, YGM provides the `ygm::io::multi_output`. This object takes a filename that a line of output must be written to and communicates the line to a specific rank that is responsible for writing to that filename.

An example of doing so is:

```
std::string output_directory{"output_dir/"};
ygm::io::multi_output mo(world, output_directory);

mo.async_write_line("file1", 14);
mo.async_write_line("file2", "this is some output");
```

One use case of this functionality is when each line of output has a timestamp, and the output lines need to be organized by the day associated with their timestamp. This behavior is provided by the `ygm::io::daily_output`, which acts the same as the *multi_output*, but all calls to `async_write_line` take a timestamp as the number of seconds since the Unix epoch instead of a filename. Files are then written to in a directory format of `year/month/day` within the output directory passed to the `ygm::io::daily_output` constructor.

4.2.1 ygm::io::line_parser

class **`line_parser`** : public ygm::container::detail::base_iteration_value<*line_parser*, std::tuple<std::string>>

Distributed text file parsing.

Public Types

using **`for_all_args`** = std::tuple<std::string>

using **`value_type`** = typename std::tuple_element<0, *for_all_args*>::type

Public Functions

inline **`line_parser`**(ygm::*comm* &comm, const std::vector<std::string> &stringpaths, bool node_local_filesystem = false, bool recursive = false)

Construct a new line parser object.

Parameters

- **`comm`** – Communicator to use for communication
- **`stringpaths`** – Vector of paths to files to read
- **`node_local_filesystem`** – True if paths are to a node-local filesystem
- **`recursive`** – True if directory traversal should be recursive

template<typename **Function**>
inline void **for_all**(*Function* fn)

Executes a user function for every line in a set of files.

Template Parameters
Function – functor type

Parameters
fn – User function to execute

inline std::string **read_first_line**()

inline void **set_skip_first_line**(bool skip_first)

inline ygm::*comm* &**comm**()

inline const ygm::*comm* &**comm**() const

inline void **for_all**(Function &&fn)

Iterates over all items in a container and executes a user-provided function object on each.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters
Function – Type of user-provided function

Parameters
fn – User-provided function

inline void **for_all**(Function &&fn) const

Const version of for_all that iterates over all items and passes them to the user function as const *.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters
Function – Type of user-provided function

Parameters
fn – User-provided function

inline void **gather**(STLContainer &gto, int rank) const

Gather all values in an STL container.

Template Parameters
STLContainer – Type of STL container to gather to

Parameters

- **gto** – Container to store results in
- **rank** – Rank to tather results on. Use -1 to gather to all ranks

inline void **gather**(STLContainer &gto) const

Gather all values in an STL container on all ranks.

Equivalent to `gather(gto, -1)`

Template Parameters
`STLContainer` – Type of STL container to gather to

Parameters
`gto` – Container to store results in

inline std::vector<*value_type*> `gather_topk`(size_t k, Compare comp = std::greater<*value_type*>()) const requires SingleItemTuple<*for_all_args*>

Gather the k "largest" values according to provided comparison function.

Template Parameters
`Compare` – Type of comparison operator

Parameters

- `k` – Number of values to gather
- `comp` – Comparison function for identifying elements to gather

Returns
vector of largest values

inline *value_type* `reduce`(MergeFunction merge) const

Perform a reduction over all items in container.

`reduce` only makes sense to use with commutative and associative functors defining merges. Otherwise, ranks will not receive the same result.

Template Parameters
`MergeFunction` – Merge functor type

Parameters
`merge` – Functor to combine pairs of items

Returns
Value from all reductions

inline void `collect`(YGMContainer &c) const

Collects all items in a new YGM container.

Template Parameters
`YGMContainer` – Container type

Parameters
`c` – Container to collect into

inline void `reduce_by_key`(MapType &map, ReductionOp reducer) const

Reduces all values in key-value pairs with matching keys.

Template Parameters

- `MapType` – Result YGM container type
- `ReductionOp` – Functor type

Parameters

- `map` – YGM container to hold result
- `reducer` – Functor for combining values

transform_proxy_value<*line_parser*, TransformFunction> **transform**(TransformFunction &&ffn)

Creates proxy that transforms items in container that are presented to user `for_all` calls.

The underlying items within the container are not modified.

```
ygm::container::bag<int> my_bag(world);
my_bag.async_insert(2);
my_bag.barrier();

my_bag.transform([](auto &val) { return 2*val; }).for_all([](const auto
&transformed_val) { YGM_ASSERT_RELEASE(val == 4);
});

my_bag.for_all([](const auto &val) { YGM_ASSERT_RELEASE(val == 2); });
```

will complete successfully.

Template Parameters

TransformFunction – functor type

Parameters

ffn – Function to transform items in container

inline flatten_proxy_value<*line_parser*> **flatten**()

Flattens STL containers of values to allow a function to be called on inner items individually.

Underlying container is not modified.

```
ygm::container::bag<std::vector<int>> my_bag(world, {{1, 2, 3}});

my_bag.flatten().for_all([](const int &nested_val) {
std::cout << "Nested value: " << nested_val << std::cout;
});
```

will print

```
Nested value: 1
Nested value: 2
Nested value: 3
```

filter_proxy_value<*line_parser*, FilterFunction> **filter**(FilterFunction &&ffn)

Filters items in a container so only allow `for_all` to execute on those that satisfy a given predicate function.

Filtered items are not removed from underlying container.

```
ygm::container::bag<int> my_bag(world, {1, 2, 3, 4});
my_bag.filter([](const auto &val) { return (val % 2) == 0;
}).for_all([](const auto &filtered_val) { YGM_ASSERT_RELEASE((filtered_val %
2) == 0);
});
```

Template Parameters
`FilterFunction` – Functor type

Parameters
`ffn` – Function used to filter items in container.

4.2.2 ygm::io::csv_parser

class **`csv_parser`** : public ygm::container::detail::base_iteration_value<*csv_parser*, std::tuple<std::vector<detail::*csv_field*>>>

Class for parsing collections of CSV files in distributed memory.

Public Types

using **`for_all_args`** = std::tuple<std::vector<detail::*csv_field*>>

using **`value_type`** = typename std::tuple_element<0, *for_all_args*>::type

Public Functions

template<typename ...**`Args`**>
inline **`csv_parser`**(*Args*&&... args)

template<typename **`Function`**>
inline void **`for_all`**(*Function* fn)

Executes a user function for every CSV record in a set of files.

Template Parameters
`Function` – functor type

Parameters
`fn` – User function to execute

inline void **`read_headers`**()

Read the header of a CSV file.

inline bool **`has_header`**(const std::string &label)

Checks for existence of a column label within headers.

Parameters
`label` – Header label to search for within headers

inline ygm::*comm* &**`comm`**()

inline const ygm::*comm* &**`comm`**() const

inline void **`for_all`**(Function &&fn)

Iterates over all items in a container and executes a user-provided function object on each.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters
`Function` – Type of user-provided function

Parameters
`fn` – User-provided function

inline void `for_all`(Function &&fn) const

Const version of for_all that iterates over all items and passes them to the user function as const *.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters
`Function` – Type of user-provided function

Parameters
`fn` – User-provided function

inline void `gather`(STLContainer &gto, int rank) const

Gather all values in an STL container.

Template Parameters
`STLContainer` – Type of STL container to gather to

Parameters

- `gto` – Container to store results in
- `rank` – Rank to tather results on. Use `-1` to gather to all ranks

inline void `gather`(STLContainer &gto) const

Gather all values in an STL container on all ranks.

Equivalent to `gather(gto, -1)`

Template Parameters
`STLContainer` – Type of STL container to gather to

Parameters
`gto` – Container to store results in

inline std::vector<*value_type*> `gather_topk`(size_t k, Compare comp = std::greater<*value_type*>()) const requires SingleItemTuple<*for_all_args*>

Gather the k “largest” values according to provided comparison function.

Template Parameters
`Compare` – Type of comparison operator

Parameters

- `k` – Number of values to gather
- `comp` – Comparison function for identifying elements to gather

Returns
vector of largest values

inline *value_type* **reduce**(MergeFunction merge) const

Perform a reduction over all items in container.

`reduce` only makes sense to use with commutative and associative functors defining merges. Otherwise, ranks will not receive the same result.

Template Parameters
MergeFunction – Merge functor type

Parameters
merge – Functor to combine pairs of items

Returns
Value from all reductions

inline void **collect**(YGMContainer &c) const

Collects all items in a new YGM container.

Template Parameters
YGMContainer – Container type

Parameters
c – Container to collect into

inline void **reduce_by_key**(MapType &map, ReductionOp reducer) const

Reduces all values in key-value pairs with matching keys.

Template Parameters

- **MapType** – Result YGM container type
- **ReductionOp** – Functor type

Parameters

- **map** – YGM container to hold result
- **reducer** – Functor for combining values

transform_proxy_value<*csv_parser*, TransformFunction> **transform**(TransformFunction &&ffn)

Creates proxy that transforms items in container that are presented to user `for_all` calls.

The underlying items within the container are not modified.

```
ygm::container::bag<int> my_bag(world);
my_bag.async_insert(2);
my_bag.barrier();

my_bag.transform([](auto &val) { return 2*val; }).for_all([](const auto
&transformed_val) { YGM_ASSERT_RELEASE(val == 4);
});

my_bag.for_all([](const auto &val) { YGM_ASSERT_RELEASE(val == 2); });
```

will complete successfully.

Template Parameters
TransformFunction – functor type

Parameters
ffn – Function to transform items in container

inline flatten_proxy_value<*csv_parser*> **flatten**()

Flattens STL containers of values to allow a function to be called on inner items individually.

Underlying container is not modified.

```
ygm::container::bag<std::vector<int>> my_bag(world, {{1, 2, 3}});

my_bag.flatten().for_all([](const int &nested_val) {
std::cout << "Nested value: " << nested_val << std::cout;
});
```

will print

```
Nested value: 1
Nested value: 2
Nested value: 3
```

filter_proxy_value<*csv_parser*, FilterFunction> **filter**(FilterFunction &&ffn)

Filters items in a container so only allow `for_all` to execute on those that satisfy a given predicate function.

Filtered items are not removed from underlying container.

```
ygm::container::bag<int> my_bag(world, {1, 2, 3, 4});
my_bag.filter([](const auto &val) { return (val % 2) == 0;
}).for_all([](const auto &filtered_val) { YGM_ASSERT_RELEASE((filtered_val %
2) == 0);
});
```

Template Parameters
FilterFunction – Functor type

Parameters
ffn – Function used to filter items in container.

4.2.3 ygm::io::ndjson_parser

class **ndjson_parser** : public ygm::container::detail::base_iteration_value<*ndjson_parser*, std::tuple<boost::json::object>>

Parser for handling collections of newline-delimited JSON files in parallel.

Public Types

using **`for_all_args`** = std::tuple<boost::json::object>

using **`value_type`** = typename std::tuple_element<0, *for_all_args*>::type

Public Functions

template<typename ...**`Args`**>
inline **`ndjson_parser`**(*Args*&&... args)

template<typename **`Function`**>
inline void **`for_all`**(*Function* fn)

Executes a user function for every CSV record in a set of files.

Template Parameters
`Function` –

Parameters
`fn` – User function to execute

inline ygm::*comm* &**`comm`**()

inline const ygm::*comm* &**`comm`**() const

inline size_t **`num_invalid_records`**()

inline void **`for_all`**(Function &&fn)

Iterates over all items in a container and executes a user-provided function object on each.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters
`Function` – Type of user-provided function

Parameters
`fn` – User-provided function

inline void **`for_all`**(Function &&fn) const

Const version of for_all that iterates over all items and passes them to the user function as const *.

The user-provided function is expected to take a single argument that is an item within the container. If the user provides a lambda as their function object, the lambda is allowed to capture.

Template Parameters
`Function` – Type of user-provided function

Parameters
`fn` – User-provided function

inline void **gather**(STLContainer &gto, int rank) const

Gather all values in an STL container.

Template Parameters

STLContainer – Type of STL container to gather to

Parameters

- **gto** – Container to store results in
- **rank** – Rank to tather results on. Use -1 to gather to all ranks

inline void **gather**(STLContainer &gto) const

Gather all values in an STL container on all ranks.

Equivalent to `gather(gto, -1)`

Template Parameters

STLContainer – Type of STL container to gather to

Parameters

gto – Container to store results in

inline std::vector<*value_type*> **gather_topk**(size_t k, Compare comp = std::greater<*value_type*>()) const requires SingleItemTuple<*for_all_args*>

Gather the k “largest” values according to provided comparison function.

Template Parameters

Compare – Type of comparison operator

Parameters

- **k** – Number of values to gather
- **comp** – Comparison function for identifying elements to gather

Returns

vector of largest values

inline *value_type* **reduce**(MergeFunction merge) const

Perform a reduction over all items in container.

`reduce` only makes sense to use with commutative and associative functors defining merges. Otherwise, ranks will not receive the same result.

Template Parameters

MergeFunction – Merge functor type

Parameters

merge – Functor to combine pairs of items

Returns

Value from all reductions

inline void **collect**(YGMContainer &c) const

Collects all items in a new YGM container.

Template Parameters

YGMContainer – Container type

Parameters

c – Container to collect into

inline void **reduce_by_key**(MapType &map, ReductionOp reducer) const

Reduces all values in key-value pairs with matching keys.

Template Parameters

- **MapType** – Result YGM container type
- **ReductionOp** – Functor type

Parameters

- **map** – YGM container to hold result
- **reducer** – Functor for combining values

transform_proxy_value<*ndjson_parser*, TransformFunction> **transform**(TransformFunction &&ffn)

Creates proxy that transforms items in container that are presented to user `for_all` calls.

The underlying items within the container are not modified.

```
ygm::container::bag<int> my_bag(world);
my_bag.async_insert(2);
my_bag.barrier();

my_bag.transform([](auto &val) { return 2*val; }).for_all([](const auto
&transformed_val) { YGM_ASSERT_RELEASE(val == 4);
});

my_bag.for_all([](const auto &val) { YGM_ASSERT_RELEASE(val == 2); });
```

will complete successfully.

Template Parameters

TransformFunction – functor type

Parameters

ffn – Function to transform items in container

inline flatten_proxy_value<*ndjson_parser*> **flatten**()

Flattens STL containers of values to allow a function to be called on inner items individually.

Underlying container is not modified.

```
ygm::container::bag<std::vector<int>> my_bag(world, {{1, 2, 3}});

my_bag.flatten().for_all([](const int &nested_val) {
std::cout << "Nested value: " << nested_val << std::cout;
});
```

will print

```
Nested value: 1
Nested value: 2
Nested value: 3
```

filter_proxy_value<*ndjson_parser*, FilterFunction> **filter**(FilterFunction &&ffn)

Filters items in a container so only allow `for_all` to execute on those that satisfy a given predicate function.

Filtered items are not removed from underlying container.

```
ygm::container::bag<int> my_bag(world, {1, 2, 3, 4});
my_bag.filter([](const auto &val) { return (val % 2) == 0;
}).for_all([](const auto &filtered_val) { YGM_ASSERT_RELEASE((filtered_val %
2) == 0);
});
```

Template Parameters

FilterFunction – Functor type

Parameters

ffn – Function used to filter items in container.

4.2.4 ygm::io::parquet_parser

class **parquet_parser**

Public Types

using **parquet_type_variant** = std::variant<std::monostate, bool, int32_t, int64_t, float, double, std::string>

Public Functions

inline **parquet_parser**(ygm::*comm* &_comm, const std::vector<std::string> &stringpaths, const bool recursive = false)

inline **~parquet_parser**()

inline const std::vector<*column_schema_type*> &**get_schema**() const

Returns a list of column schema (simpler version). The order of the schema is the same as the order of Parquet column indices (ascending order). This function assumes that all files have the same schema. Returns an empty vector if there is no file the rank can read.

inline const std::string &**schema_to_string**() const

template<typename Function> inline requires std::invocable< Function, const std::vector< parquet_type_variant > & > void for_all (Function fn, const size_t num_rows=std::numeric_limits< size_t >::max())

Read all rows and call the function for each row.

Parameters

- **fn** – A function to call for every row. Expected signature is void(const std::vector<parquet_type_variant>&). The value of an unsupported column is set to std::monostate.

- **`num_rows`** – Max number of rows the rank to read.

`template<typename Function> inline requires std::invocable< Function, const std::vector< parquet_type_variant > & > void for_all (const std::vector< std::string > &colum Function fn, const size_t num_rows=std::numeric_limits< size_t >::max())`

for_all(), read only the specified columns.

inline std::optional<std::vector<*parquet_type_variant*>> **`peek`**()

Return the first row assigned to the rank. Return nullopt if no row was assgined.

inline size_t **`num_files`**() const

Return the total number of files.

inline size_t **`num_rows`**() const

Return the number of rows in all files.

struct **`column_schema_type`**

Public Members

detail::parquet_data_type **`type`**

std::string **`name`**

bool **`unsupported`** = {false}

4.2.5 ygm::io::multi_output

template<typename **`Partitioner`** = ygm::container::detail::old_hash_partitioner<std::string>>

class **`multi_output`**

Class for writing output to multiple named files in distributed memory.

Template Parameters

`Partitioner` – Type used to assign filenames to ranks for writing

Public Types

using **`self_type`** = *multi_output*<*Partitioner*>

Public Functions

inline **multi_output**(ygm::*comm* &comm, std::string filename_prefix, size_t buffer_length = 1024 * 1024, bool append = false)

Construct a *multi_output* object.

filename_prefix is assumed to be a directory name and has a "/" appended if not already present to force it to be a directory

Parameters

- **comm** – Communicator to use for communication
- **filename_prefix** – Prefix used when creating filenames
- **buffer_length** – Length of buffers to use before writing
- **append** – If false, existing files are overwritten. Otherwise, output is appended to existing files.

inline **~multi_output**()

template<typename ...**Args**>
inline void **async_write_line**(const std::string &subpath, *Args*&&... args)

Write a line of output.

Template Parameters

Args... – Variadic types of output

Parameters

- **subpath** – Filename to append to filename_prefix mutli_output is created with when creating full output path
- **args...** – Variadic arguments to write to output file

inline ygm::*comm* &**comm**()

4.2.6 ygm::io::daily_output

template<typename **Partitioner** = ygm::container::detail::old_hash_partitioner<std::string>>

class **daily_output**

Class for writing output to a file for each day based on a timestamp provided at the time of writing.

Template Parameters

Partitioner – Type used to assign filenames to ranks for writing

Public Types

using **self_type** = *daily_output*<*Partitioner*>

Public Functions

inline **daily_output**(ygm::*comm* &comm, const std::string &filename_prefix, size_t buffer_length = 1024 * 1024, bool append = false)

Construct a *daily_output* object.

Parameters

- **comm** – Communicator to use for communication
- **filename_prefix** – Prefix used when creating filenames
- **buffer_length** – Length of buffers to use before writing
- **append** – If false, existing files are overwritten. Otherwise, output is appended to existing files.

template<typename ...**Args**>
inline void **async_write_line**(const uint64_t timestamp, *Args*&&... args)

Write a line of output.

Template Parameters

Args... – Variadic types of output

Parameters

- **timestamp** – Linux timestamp associated to use when assigning output to a file
- **args...** – Variadic arguments to write to output file

CHAPTER

FIVE

YGM::UTILITY MODULE REFERENCE

The utility namespace contains multiple components that often helpful when using YGM, but may not be necessary to use. Their uses include tracking the performance of YGM, getting easy access to basic functionality built on *MPI_COMM_WORLD*, and sending messages using YGM that contain specialized data types. The headers containing this functionality can be safely included in user programs, but is often already included in other YGM headers because they can be used within YGM.

5.1 ygm::utility::timer

The *ygm::utility::timer* class starts a very simple timer using *MPI_Wtime*. It includes *elapsed()* and *reset()* methods for checking the time since the timer has been started and resetting the start time of a timer, respectively.

Typical use of the *ygm::utility::timer* is:

```
ygm::utility::timer t{};
{
  // Do stuff
}
world.barrier();
world.cout0("Time: ", t.elapsed());
```

5.2 ygm::utility::progress_indicator

The *ygm::utility::progress_indicator* asynchronously tracks progress through a calculation across all processes, with each process periodically sending updates that are printed by rank 0. The *progress_indicator* prints the total number of work items completed and the rate at which they are completing.

The *async_inc()* method of the *progress_indicator* is used to locally indicate when work is progressing. This call will begin a nonblocking reduction when enough work has been completed to collect the output for printing. An internal *progress_indicator::options* class is used to control the message for printing and the frequency with which reductions and printing occurs.

Typical use of the *ygm::utility::timer* is:

```
ygm::utilijty::progress_indicator prog(world, {.update_freq = 10, .message = "Doing stuff
↪"});
for (int i=0; i<1000; ++i) {
 prog.async_inc();
 // Do work
```

(continues on next page)

(continued from previous page)

```
}
prog.complete();
world.barrier();
```

5.3 Global World Functionality

YGM provides the following functions for basic interactions with *MPI_COMM_WORLD* that can be done from anywhere within a YGM program:

- *ygm::wrank()* - returns the current process's rank
- *ygm::wrank0()* - returns a boolean indicating whether the current rank is rank 0 or not
- *ygm::wsize()* - returns the number of ranks in *MPI_COMM_WORLD*
- *ygm::wcout0()* - prints output to *std::cout* from only rank 0
- *ygm::wcerr0()* - same as *ygm::wcout0()* but provides *std::cerr* access for just rank 0
- *ygm::wcout()* - prints output to *std::cout* from the current rank with its rank prepended
- *ygm::wcerr()* - same as *ygm::wcout()* but prints to *std::cerr*

The printing functionality can be used either to get access to an output stream for printing or as a *print()* type of function, that is *ygm::wcout0() << "Printint output"* and *ygm::wcout0("Printing output")* will produce the same output.

5.4 Asserts

assert.hpp provides a small number of assert macros:

- *YGM_ASSERT_MPI* - used for wrapping MPI calls to detect when MPI does not return *MPI_SUCCESS*
- *YGM_ASSERT_DEBUG* - same functionality as *assert*
- *YGM_ASSERT_RELEASE* - assert statement that is triggered even if *NDEBUG* is defined

5.5 Specialized Serialization Functions

A number of headers are provided for serialization of datatypes for communication through YGM:

- *boost_*.hpp* - serialization for various Boost types

5.6 Utility Class Documentation

5.6.1 ygm::utility::timer

class **`timer`**

Simple timer class using `MPI_Wtime()`

Public Functions

inline **timer**()

inline double **elapsed**()

Get time since timer creation or last *reset()*

Returns

Elapsed time

inline void **reset**()

Restart timer.

5.6.2 ygm::utility::progress_indicator

class **progress_indicator**

Simple progress indicator class.

```
ygm::progress_indicator prog(world);
for (size_t i = 0; i < 1000; ++i) {
  prog.async_inc();
  std::this_thread::sleep_for(std::chrono::milliseconds(1));
}
prog.complete();
world.barrier();
```

Public Functions

inline **progress_indicator**(ygm::*comm* &comm, const *options* &opts)

inline **~progress_indicator**()

inline void **async_inc**(size_t i = 1)

Asynchronously update progress from local rank.

Parameters

i – amount to increment progress

inline void **complete**()

Complete the progress indicator.

This is a collective function and should be called prior to a barrier()

struct **options**

Public Members

size_t **update_freq** = 10

How frequently to attempt global reduction.

std::string **message** = "Progress"

Message header to print.

CHAPTER

SIX

DOCUMENTS FOR YGM DEVELOPERS

6.1 Developing YGM

This page contains information for YGM developers.

6.1.1 Build Read the Docs (RTD)

Here is how to build RTD document using Sphinx on your machine.

Listing 1: How to build RTD docs locally

```
# Install required software
brew install doxygen graphviz sphinx-doc
pip install breathe sphinx_rtd_theme

# Set PATH and PYTHONPATH, if needed
# For example:
# export PATH="/opt/homebrew/opt/sphinx-doc/bin:${PATH}"
# export PYTHONPATH="/path/to/python/site-packages:${PYTHONPATH}"

git clone https://github.com/LLNL/ygm.git
cd ygm
mkdir build && cd build

# Run CMake
cmake ../ -DYGM_RTD_ONLY=ON

# Generate Read the Docs documents using Sphinx
# This command runs Doxygen to generate XML files
# before Sphinx automatically
make sphinx
# Open the following file using a web browser
open docs/rtd/sphinx/index.html

# For running doxygen only
make doxygen
# open the following file using a web browser
open docs/html/index.html
```

Rerunning Build Command

Depending on what files are modified, one may need to rerun the CMake command and/or `make sphinx`. For instance:

- Require running the CMake command and `make sphinx`:
  - Adding new RTD-related files, including configuration and .rst files
  - Modifying CMake files
- Require running only `make sphinx`
  - **Existing** files (except CMake) are modified

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX