GASPI: Global Address Space Programming Interface

Specification of a PGAS API for communication

Version 16.1

September 15, 2016

Contents

1 Introduction to GASPI

1.1 Overview and Goals

GASPI stands for Global Address Space Programming Interface and is a Partitioned Global Address Space (PGAS) API. It aims at extreme scalability, high flexibility and failure tolerance for parallel computing environments. GASPI aims to initiate a paradigm shift from bulk-synchronous two-sided communication patterns towards an asynchronous communication and execution model. To that end GASPI leverages remote completion and one-sided RDMA driven communication in a Partitioned Global Address Space.

GASPI is neither a new language (like Chapel from Cray), nor an extension to a language (like Co-Array Fortran or UPC). Instead—very much in the spirit of MPI—it complements existing languages like C/C++ or Fortran with a PGAS API which enables the application to leverage the concept of the Partitioned Global Adress Space. GASPI is not limited to a single memory model, but rather provides configurable RDMA PGAS memory segments. GASPI allows application developers to map the memory heterogeneity of a modern supercomputer node to these PGAS segments. As an example GASPI allows users to map the main memory of a GPGPU or Xeon Phi to a specific segment, to configure a GASPI segment per memory controller in a CC-NUMA system or to map non-volatile RAM to a specific segment. All these segments can directly read and write from/to each other - within the node and across all nodes. GASPI is failure tolerant in the sense that it provides timeout mechanisms for all non-local procedures, failure detection and the possibility to adapt to shrinking or growing node sets.

1.2 History

The GASPI specification originates from the PGAS API of the Fraunhofer ITWM (Fraunhofer Virtual Machine, FVM), which has been developed since 2005. Starting from 2007 this PGAS API has evolved into a robust commercial product (called GPI) which is used in the industry projects of the Fraunhofer ITWM. GPI offers a highly efficient and scalable programming model for Partitioned Global Address Spaces and has replaced MPI completely at Fraunhofer ITWM. In 2011 the partners of Fraunhofer ITWM, Fraunhofer SCAI, TUD, T-Systems SfR, DLR, KIT, FZJ, DWD and Scapos have initiated and launched the GASPI project to define a novel specification for a PGAS API (GASPI, based on GPI) and to make this novel GASPI specification a reliable, scalable and universal tool for the HPC community.

1.3 Design goals

GASPI has been designed with the following goals in mind:

- Extreme scalability.

- Efficient one sided asynchronous remote read/write operations based on remote completion.
- Multi-segment support to support e. g. heterogeneous systems and NUMA-pinning.
- Dynamic allocation of segments.
- Timeout mechanisms to allow failure tolerant programming.
- Asynchronous collective operations for groups of processes.
- Flexibility in the number of message queues, the queue sizes, atomic operations etc.
- A maximum freedom to implementors, where details are left to the implementation.
- A strong standard library which takes care of convenience procedures and cosmetics. The specification should be simple and solid.

2 GASPI terms and conventions

This section describes notational terms and conventions used throughout the GASPI document.

2.1 Naming Conventions

All procedures are named in accordance with the following convention. The procedures have `gaspi_` as a prefix. The prefix is followed by the operation name.

2.2 Procedure specification

GASPI has adopted the procedure specification of MPI. Similar to the MPI standard, procedures in GASPI hence are first specified using a language independent notation. Immediately below this, the arguments of the procedure are given and marked as *in* or *out*. The meanings of these are:

- the call uses but does not update an argument marked *in*. For the C procedures these arguments are const-correct.
- the call may update an argument marked *out*.

Similar to MPI, in GASPI the passing of aliased procedure parameters results in undefined behavior.

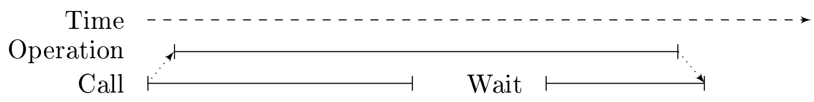
Below the procedure arguments the ANSI C version of the function is shown, and below this, a version of the same function is shown for Fortran 2003. For the latter the corresponding definitions and derived types have to be include via

```
use GASPI_C_BINDING
```

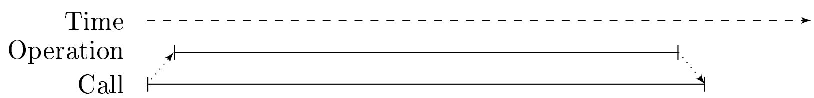
2.3 Semantic terms

The following semantic terms are used throughout the document:

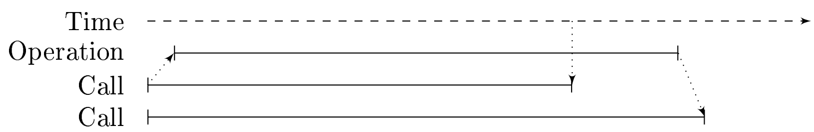
non-blocking A procedure is non-blocking if the procedure may return before the operation completes.



blocking A procedure is blocking if the procedure only returns after the operation has completed.



time-based blocking A procedure is time-based blocking if the procedure may return after the operation completes or after a given timeout has been reached. A corresponding return value is used to distinguish between the two cases.



local A procedure is local if completion of the procedure depends only on the local executing GASPI process.

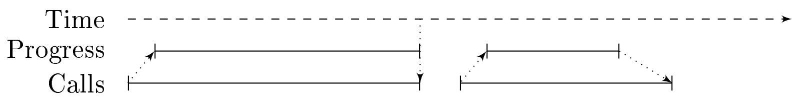
non-local A procedure is non-local if completion of the operation may depend on the existence (and execution) of a remote GASPI process

collective A procedure is collective if all processes in a process group need to invoke the procedure. A collective call may or may not be synchronising.

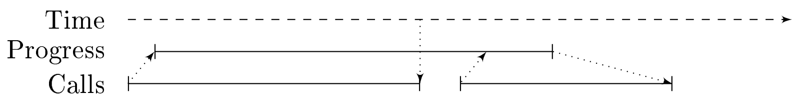
predefined A predefined type is a datatype with a predefined constant name.

timeout A timeout is a mechanism required by procedures that might block (see blocking above). Timeout here is defined as the maximum time (in milliseconds) a called procedure will wait for outstanding communication from other processes. The special value 0 (defined as `GASPI_TEST`) indicates that the procedure will complete a portion of its work, if possible. The procedure subsequently returns the current status without waiting for data from other processes (non-blocking). On the other hand the special value -1 (defined as `GASPI_BLOCK`) instructs the procedure to wait indefinitely (blocking). A number greater than 0 indicates the maximum time the procedure will wait for data from other ranks (time-based blocking). The timeouts hence are soft: The timeout value n does not imply that the called procedure will return after n milliseconds. It just means that the procedure should wait for at most n milliseconds for data from other processes.

synchronous A procedure is called synchronous if progress towards completion only is achieved as long as the application is inside (executing) the procedure.



asynchronous A procedure is called asynchronous if progress towards completion may be achieved after the procedure exits.



Please note that some of the semantic terms are not exclusive. Some of them do overlap. According to the definition, a collective procedure may also be a local procedure. Furthermore, a blocking procedure is per definition also a synchronous procedure; the reverse statement is not true.

2.4 Examples

The examples in this document are for illustration purposes only. They are not intended to specify the semantics.

3 GASPI concepts

3.1 Introduction and overview

In this section, the basic GASPI concepts are introduced. A more detailed description with the corresponding procedure specifications can be found in the subsequent topic-specific sections.

GASPI is a communication API that implements a Partitioned Global Address Space (*PGAS*) model. Each GASPI process may host parts (called segments) of the global address space. A local segment can be accessed with standard load/store operations and remote segments can be accessed by every thread of every GASPI process using the GASPI read and write operations.

GASPI was designed with remote direct memory access (RDMA) in mind. A network infrastructure that supports RDMA guarantees asynchronous and one-sided communication operations without involving the CPU. This is one of the main requirements for high scalability which results from interference free communication, e. g. from overlapping communication with computation.

3.2 GASPI processes

GASPI preserves the concept of ranks. Each GASPI process receives a unique rank that identifies it during its runtime.

3.3 GASPI groups

A group is a subset of all processes. The group members have common collective operations. A collective operation is then restricted to the processes forming the group.

3.4 GASPI segments

Modern hardware typically involves a hierarchy of memory with respect to the bandwidth and latencies of read and write accesses. Within that hierarchy are non-uniform memory access (*NUMA*) partitions, solid state devices (*SSD*s), graphical processing unit (*GPU*) memory or many integrated cores (*MIC*) memory. The GASPI memory segments are supposed to map this variety of hardware layers to the software layer. In the spirit of the PGAS approach, these GASPI segments may be globally accessible from every thread of every GASPI process. GASPI segments can also be used to leverage different memory models within a single application or to even run different applications in a single Partitioned Global Address Space.

3.5 GASPI one-sided communication

One-sided asynchronous communication is the basic communication mechanism provided by GASPI. The one-sided communication comes in two flavors. There are read and write operations from and into the Partitioned Global Address Space. For the write operations GASPI makes use of the concept of remote completion in the form of so-called notifications. One-sided operations are non-blocking and asynchronous, allowing the program to continue its execution along the data transfer. The actual data transfer is managed by the underlying network infrastructure.

3.6 GASPI queues

GASPI offers the possibility to use different queues to handle the communication requests. The requests can be submitted to one of the supported queues. These queues allow more scalability and can be used as channels for different types of requests where similar types of requests are queued and then get synchronised together but independently from the other ones (separation of concerns). The specification guarantees fairness of transfers posted to different queues, i. e. no queue should see its communication requests delayed indefinitely.

Listing 1: Allgather with one-sided writes.

```
let nProc be the number of processes;
let iProc be the unique id of this process;
let src be the data to be distributed;
let dst be an array storing the destination addresses;

foreach process p in [0,nProc):
  write src into dst[p][iProc];
  //             ^^^^^^
  //             | remote address if p != iProc

wait for the completion of the writes;

barrier;
// the writes of all processes are completed
```

3.7 GASPI passive communication

Passive communication has a two-sided semantic, where there is a matching receive operation to a send request. Passive communication aims at communication patterns where the sender is unknown (i. e. it can be any process from the receiver perspective) but there is potentially the need for synchronisation between different processes.

The receive operation is a blocking call that has as low interference as possible (e. g. consumes no CPU cycles) and is ideally woken up by the network layer. This passive communication allows for fair distributed updates of globally shared parts of data.

3.8 GASPI global atomics

GASPI provides atomic operations for integral types, i. e. such variables can be manipulated atomically without fear of preemption causing corruption. There are two basic atomic operations: `fetch_and_add` and `compare_and_swap`. The values can be used as global shared variables and to synchronise processes or events.

The specification guarantees fairness, i. e. no process should see its atomic operation delayed indefinitely.

Listing 2: Dynamic work distribution: Clients atomically fetch a packet id and increment the value.

```
do
{
  packet := fetch_and_add (1);
```

```
  // increment the value by one, return the old value

  if (packet < packet_max): process (packet);
}
while (packet < packet_max);
```

3.9 GASPI timeouts

Failure tolerant parallel programs necessitate non-blocking communication calls. Hence, GASPI provides a timeout mechanism for all potentially blocking procedures.

Timeouts for procedures are specified in milliseconds. `GASPI_BLOCK` is a predefined timeout value which blocks the procedure call until completion. This value should not be used in failure tolerant programs, as it can block for an indefinitely amount of time in case of an error.

`GASPI_TEST` is another predefined timeout value which blocks the procedure for the shortest time possible, i. e. the time in which the procedure call processes a portion of its work, if possible.

Examples:

Listing 3: Blocks until the communication queue is empty and may block indefinitely in case of a failure.

```
WAIT (..., GASPI_BLOCK);
```

Listing 4: Just check if the operation has completed and return as soon as possible.

```
WAIT (..., GASPI_TEST);
```

Listing 5: Blocks until the queue is empty or more than 10 milliseconds have passed since wait has been called.

```
WAIT (..., 10);
```

3.10 GASPI collective communication

Collective communication is communication which involves a group of GASPI processes. It is collective only for that group.

Collective operations can be either synchronous or asnychronous. Synchronous implies that progress is achieved only as long as the application is inside of the call. The call itself, however, may be interrupted by a timeout. The operation is then continued in the next call of the procedure. This implies that a collective operation may involve several procedure calls until completion.

Collective operations are exclusive per group, i. e. only one collective operation of a specific type can run at a given time for a given group. For example, two allreduce for one group cannot run at the same time; however, an allreduce operation and a barrier can run at the same time.

> *Implementor advice:* GASPI does not regulate whether individual collective operations should internally be handled synchronously or asynchronously, however: GASPI aims at an efficient, low-overhead programming model. If asynchronous operation is supported, it should leverage external network-resources, rather than consuming CPU cycles. ⌟

GASPI supports the following collective operations: barriers, reductions with predefined operations, reductions with user defined operations.

Collective operations are/will be synchronized independently from the operations on the communication queues.

3.11 GASPI return values

GASPI procedures have three general return values:

`GASPI_SUCCESS` implies that the procedure has completed successfully.

`GASPI_TIMEOUT` implies that the procedure could not complete in the given period of time. This does not necessitate an error. The procedure has to be invoked subsequently in order to fully complete the operation.

`GASPI_ERROR` implies that the procedure has terminated due to an error. There are no predefined error values specifying the detailed cause of an error. `gaspi_error_message` translates the error code into a human readable format.

> *Implementor advice:* An implementation may provide specific error values. All error codes in the range $[-1, \ldots, -999]$ are reserved and must not be used. If there are predefined error codes, each of the return codes must have a corresponding error message. ⌟

Additionally, each process has a state vector that contains the health state for all processes. The state vector is set after non-local operations and can be used to detect failures on remote processes.

4 GASPI definitions

4.1 Types

```
gaspi_rank_t
```

The GASPI *rank type.* ⌟

```
gaspi_segment_id_t
```

The GASPI *memory segment ID type.* ⌟

```
gaspi_offset_t
```

The GASPI *offset type. Offsets are measured relative to the beginning of a memory segment in units of bytes.* ⌟

```
gaspi_size_t
```

The GASPI *size type. Sizes are measured in units of bytes.* ⌟

```
gaspi_queue_id_t
```

The GASPI *queue ID type.* ⌟

```
gaspi_notification_t
```

The GASPI *notification type.* ⌟

> *Implementor advice:* The sum of the sizes of `gaspi_notification_t` and `gaspi_tag_t` should be at most 8 bytes in order to allow for Infiniband specific optimizations. ⌟

```
gaspi_notification_id_t
```

The GASPI *notification ID type.* ⌟

> *Implementor advice:* The sum of the sizes of `gaspi_notification_t` should be at most 8 bytes in order to allow for Infiniband specific optimizations. ⌟

```
gaspi_atomic_value_t
```

The GASPI *global atomic value type. An atomic value is unsigned and its maximum value can be queried using* `gaspi_atomic_max`*.* ⌟

```
gaspi_return_t
```

The GASPI *return value type.* ⌟

```
vector<gaspi_return_t>
```

`gaspi_returns_t`

The vector type with return codes for individual processes. The length of the vector equals the number of processes in the GASPI *program.* ⌟

```
gaspi_timeout_t
```

The GASPI *timeout type.* ⌟

`gaspi_number_t`

A type that is used to count elements. That could be numbers of queues as well as the size of individual queues. ⌟

`gaspi_group_t`

The GASPI *group type.* ⌟

`gaspi_pointer_t`

A type that can point to some (area of) memory. ⌟

`gaspi_const_pointer_t`

A type that can point to some (area of) memory that cannot be modified using this pointer. ⌟

`gaspi_memory_description_t`

The GASPI *memory description type used to describe properties of user provided memory.* ⌟

> *Implementor advice:* The intention of `gaspi_memory_description_t` is to describe properties of memory that is provided by the application, e.g. `MEMORY_GPU` or `MEMORY_HOST` might be relevant to an implementation. ⌟

`gaspi_alloc_t`

The GASPI *allocation policy type.* ⌟

`gaspi_network_t`

The GASPI *network infrastructure type.* ⌟

`gaspi_string_t`

The GASPI *constant string type.* ⌟

`gaspi_statistic_counter_t`

The GASPI *statistic counter type.* ⌟

4.2 Constants

4.2.1 Timeout values

`GASPI_BLOCK`

`GASPI_BLOCK` *is a timeout value which blocks a procedure call until completion.* ⌟

`GASPI_TEST`

`GASPI_TEST` *is a timeout value which blocks a procedure call for the shortest time possible.* ⌟

4.2.2 Function return values

`GASPI_SUCCESS`

GASPI_SUCCESS is returned if a procedure call is completed successfully. ⌟

`GASPI_TIMEOUT`

GASPI_TIMEOUT is returned if a procedure call ran into a timeout. ⌟

`GASPI_ERROR`

GASPI_ERROR is returned if a procedure call finished with an error. ⌟

4.2.3 State vector states

GASPI_STATE_HEALTHY

GASPI_STATE_HEALTHY implies that a remote GASPI *process is healthy and communication is possible.* ⌟

GASPI_STATE_CORRUPT

GASPI_STATE_CORRUPT implies that the remote GASPI *process is corrupted and communication is impossible.* ⌟

4.2.4 Allocation policies

`GASPI_ALLOC_DEFAULT`

The `GASPI_ALLOC_DEFAULT` *policy uses the operating systems default memory allocation policy.* ⌟

> *Implementor advice:* A GASPI implementation is free to provide additional allocation policies. ⌟

4.2.5 Statistics interface

A GASPI implementation is free to define constants of the type `gaspi_statistic_counter_t` for specific statistics.

5 Execution model

5.1 Introduction and overview

GASPI allows both SPMD (Single Program, Multiple Data) and MPMD (Multiple Program, Multiple Data) style program execution. Hence, either a single program or different programs can be started on the computational units. How a GASPI application is started and initialized is implementation specific.

A rank is attributed to each created process. Ranks are a central aspect as they allow applications to identify processes and therefore allow to distribute work among the processes.

Furthermore, GASPI provides segments. Segments are globally accessible memory regions. In general, the execution of a GASPI process can be considered as split into several consecutive phases:

- **Setup (optional)**

  Setting up configuration parameters

  Performing environment checks

- **Initialization**

  Initialization of the runtime environment

  Creation of segments or groups (optional)

- **Working (optional)**

  Communication calls

  Collective operations

  Atomic operations

- **Shutdown**

  Cleanup of communication infrastructure

In the **setup** phase, the application may retrieve and modify the GASPI configuration structure (see Sect. 5.2.1) determining the GASPI runtime behavior. Optionally (but advisable), the application can perform environment checks (see Sect. 13) to make sure the application can be started safely and correctly.

In the **initialization** phase, the GASPI runtime environment is set up in accordance with the parameters of the configuration structure by invocation of the initialization procedure. The initialization procedure is called before any other functionality, with the exception of pre-initialization routines for environment checking and declaration and retrieval of configuration parameters. After the initialization routine has been called, an optional step to perform is the creation of one or more segments and the creation of one or more groups. Segments are contiguous blocks of memory that may be accessed globally by all processes and where global data should be placed.

After the initialization, the application can proceed with its **working** phase and use the functionalities of GASPI (communication, collectives, atomic operations, etc.).

The application should call the **shutdown** procedure (see Sect. 5.3.4) before it is terminated so that all resources and the communication infrastructure is cleaned up.

The entire set of execution phases define the GASPI life cycle. In principle, several life cycles can be invoked in one GASPI program.

Calling a routine in an execution phase in which it is not supposed to be executed in results in undefined behavior.

5.2 Process configuration

5.2.1 GASPI configuration structure

The GASPI configuration structure describes the configuration parameters which influence the GASPI runtime behavior.

Please note, that for simplicity of notation this is a C-style definition. In bindings to other languages corresponding definitions will be used.

Listing 6: GASPI configuration structure.

```
typedef struct {
  // maximum number of groups
  gaspi_number_t     group_max;

  // maximum number of segments
  gaspi_number_t     segment_max

  // one-sided comm parameter
  gaspi_number_t     queue_num;
  gaspi_number_t     queue_size_max;
  gaspi_size_t       transfer_size_max;

  // notification parameter
  gaspi_number_t     notification_num;

  // passive comm parameter
  gaspi_number_t     passive_queue_size_max;
  gaspi_size_t       passive_transfer_size_max;

  // collective comm parameter
```

```
  gaspi_size_t       allreduce_buf_size;
  gaspi_number_t     allreduce_elem_max;

  // network selection parameter
  gaspi_network_t    network;

  // communication infrastructure build up notification
  gaspi_number_t       build_infrastructure;

  void *             user_defined;
} gaspi_config_t;
```

The definition of each of the configuration structure fields is as follows:

group_max the desired maximum number of permissible groups per process. There is a hardware/implementation dependent maximum.

segment_max the desired number of maximally permissible segments per GASPI process. There is a hardware/implementation dependent maximum.

queue_num the desired number of one-sided communication queues to be created. There is a hardware/implementation dependent maximum.

queue_size_max the desired number of simultaneously allowed on-going requests on a one-sided communication queue. There is a hardware/implementation dependent maximum.

transfer_size_max the desired maximum size of a single data transfer in the one-sided communication channel. There is a hardware/implementation dependent maximum.

notification_num the desired number of internal notification buffers for weak synchronisation to be created. There is a hardware/implementation dependent maximum.

passive_queue_size_max the desired number of simultaneously allowed on-going requests on the passive communication queue. There is a hardware/implementation dependent maximum.

passive_transfer_size_max the desired maximum size of a single data transfer in the passive communication channel. There is a hardware/implementation dependent maximum.

allreduce_elem_max the maximum number of elements in `gaspi_allreduce`. There is a hardware/implementation dependent maximum.

allreduce_buf_size the size of the internal buffer of `gaspi_allreduce_user`. There is a hardware/implementation dependent maximum.

network the network type to be used.

build_infrastructure indicates whether the communication infrastructure should be built up at startup time. The default value is true.

user_defined some user defined information that is application / implementation dependent.

The default configuration structure can be retrieved by `gaspi_config_get`. Its default values are implementation dependent. If some of the parameters are set by the program and assigned with `gaspi_config_set`, the requested values are just proposals. Depending on the underlying hardware capabilities, the implementation is allowed to overrule these proposals. `gaspi_config_set` has to be used in order to commit modifications of the configuration structure before the initialization routine is invoked. The actual values of the parameters can be retrieved by the corresponding GASPI getter routines (see Sect. 12) after the successful program initialization. The values of the configuration structure parameters need to be the same on all GASPI processes.

The user has the possibility to set the values on her own or leave the default values. Each field (where applicable) also has a maximum value to avoid user errors that might lead to too much instability or scalability problems (for example, the number of queues).

5.2.2 gaspi_config_get

The `gaspi_config_get` procedure is a *synchronous local blocking* procedure which retrieves the default configuration structure.

```
GASPI_CONFIG_GET ( config )
```

Parameter:

(out) config: the default configuration

```
gaspi_return_t
gaspi_config_get ( gaspi_config_t *config )
```

```
function gaspi_config_get(config) &
&   result( res ) bind(C, name="gaspi_config_get")
  type(gaspi_config_t) :: config
  integer(gaspi_return_t) :: res
end function gaspi_config_get
```

Execution phase:

Setup

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, *config* represents the default configuration.

In case of error, the return value is GASPI_ERROR.

5.2.3 gaspi_config_set

The gaspi_config_set procedure is a *synchronous local blocking* procedure which sets the configuration structure for process initialization.

```
GASPI_CONFIG_SET ( config )
```

Parameter:

(in) config: the configuration structure to be set

```
gaspi_return_t
gaspi_config_set ( gaspi_config_t const config )
```

```
function gaspi_config_set(new_config) &
&   result( res ) bind(C, name="gaspi_config_set")
  type(gaspi_config_t), value :: new_config
  integer(gaspi_return_t) :: res
end function gaspi_config_set
```

Execution phase:

Setup

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, the runtime parameters for the GASPI process initialization are set in accordance with parameters of *config*.

In case of error, the return value is GASPI_ERROR.

5.3 Process management calls

5.3.1 gaspi_proc_init

gaspi_proc_init implements the GASPI initialization of the application. It is a *non-local synchronous time-based blocking* procedure.

```
GASPI_PROC_INIT ( timeout )
```

Parameter:

(in) timeout: the timeout

```
gaspi_return_t
gaspi_proc_init ( gaspi_timeout_t const timeout )
```

```
function gaspi_proc_init(timeout_ms) &
&         result( res ) bind(C, name="gaspi_proc_init")
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_proc_init
```

Execution phase:

Initialization

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

The explicit start of a GASPI process or launch from command line is not specified. This is implementation dependent.

However, it is anticipated that `gaspi_proc_init` has information about the list of hosts on which the entire GASPI application is running either by environment variables, a command line argument, a daemon or some other mechanism. The actual transfer of knowledge is implementation dependent.

`gaspi_proc_init` registers a given process at the other remote GASPI processes and sets the corresponding entries in the state vector to a healthy state. If the parameter *build_infrastructure* in the configuration structure is set, also the communication infrastructure for passive and one-sided communication to all of the other processes is setup. Otherwise, there is no set up of communication infrastructure during the initialization. A rank is assigned to the given GASPI process in accordance with the position of the host in the list. The GASPI process running on the first host in the list has rank zero. The GASPI process running on the second host in the list has rank one and so on.

In case of a node failure, a GASPI process can be started on a new host, freshly allocated or selected from a set of pre-allocated spare hosts, by providing the list of machines in which the failed node is substituted by the new host. The new GASPI process then has the rank of the GASPI process which has been running on the failed node.

In case of the subsequent start of additional GASPI processes, the newly started GASPI process registers with the other remote GASPI processes. Note, that a subsequent change of the number of running GASPI processes invalidates `GASPI_`

GROUP_ALL for the old running processes. Also the return value of gaspi_proc_num is changed.

The configuration structure should be created and modified by the application before calling the gaspi_proc_init procedure.

After successful procedure completion, gaspi_proc_init returns GASPI_SUCCESS and it guarantees that the application has been started on all hosts. In case that the *build_infrastructure* is set, return value GASPI_SUCCESS also implies that the communication infrastructure is up and ready to be used.

In case the application could not be initialized in line with the timeout parameter, the return value is GASPI_TIMEOUT. The application has not been initialized yet. A subsequent invocation is required to completely initialize the application.

In case of error, the return value is GASPI_ERROR. The application is not initialized.

> *Implementor advice:* Calling gaspi_proc_init with an enabled parameter *build_infrastructure* is semantically equivalent to calling gaspi_proc_init with a disabled parameter *build_infrastructure* and subsequent calls to gaspi_connect in which an all-to-all connection is established. ⌟

> *User advice:* For resource critical applications, it is recommended to disable the parameter *build_infrastructure* in the configuration structure. ⌟

> *User advice:* A successful procedure completion does not mean that any communication or collective operation can already be used. Connections might need to be established. A segment has to be allocated for passive communication capabilities. If one-sided communication is supposed to be used, than the segment has to be registered in addition. If collective operations are needed, a group has to be created and committed. ⌟

5.3.2 gaspi_proc_num

The total number of GASPI processes started, can be retrieved by gaspi_proc_num. This is a *local synchronous blocking* procedure.

```
GASPI_PROC_NUM ( proc_num )
```

Parameter:

(out) proc_num: the total number of GASPI processes

```
gaspi_return_t
gaspi_proc_num ( gaspi_rank_t *proc_num )
```

```
function gaspi_proc_num(proc_num) &
&         result( res ) bind(C, name="gaspi_proc_num")
  integer(gaspi_rank_t) :: proc_num
  integer(gaspi_return_t) :: res
end function gaspi_proc_num
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

If successful, the return value is `GASPI_SUCCESS` and `gaspi_proc_num` retrieves the total number of processes that have been initialized and places this number in the *proc_num*.

In case of error, the return value is `GASPI_ERROR` and the value of *proc_num* is undefined.

5.3.3 `gaspi_proc_rank`

A rank identifies a GASPI process. The rank of a process lies in the interval $[0, P)$ where P can be retrieved through `gaspi_proc_num`. Each process has a unique rank associated with it. The rank of the invoking GASPI process can be retrieved by `gaspi_proc_rank`. It is a *local synchronous blocking* procedure.

```
GASPI_PROC_RANK ( rank )
```

Parameter:

(out) rank: the rank of the calling GASPI process.

```
gaspi_return_t
gaspi_proc_rank ( gaspi_rank_t *rank )
```

```
function gaspi_proc_rank(rank) &
&         result( res ) bind(C, name="gaspi_proc_rank")
  integer(gaspi_rank_t) :: rank
  integer(gaspi_return_t) :: res
end function gaspi_proc_rank
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

gaspi_proc_rank retrieves, if successful, the rank of the calling process, placing it in the parameter *rank* and returning GASPI_SUCCESS.

In case of error, the return value is GASPI_ERROR and the value of the *rank* is undefined.

5.3.4 gaspi_proc_term

The shutdown procedure gaspi_proc_term is a *synchronous non-local time-based blocking* operation that releases resources and performs the required clean-up. There is no definition in the specification of a verification of a healthy global state (i. e. all processes terminated correctly).

After a shutdown call on a given GASPI process, it is undefined behavior if another GASPI process tries to use any non-local GASPI functionality involving that process.

```
GASPI_PROC_TERM ( timeout )
```

Parameter:

(in) timeout: the timeout

```
gaspi_return_t
gaspi_proc_term ( gaspi_timeout_t timeout )
```

```
function gaspi_proc_term(timeout_ms) &
&         result( res ) bind(C, name="gaspi_proc_term")
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_proc_term
```

Execution phase:

Shutdown

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

In case of successful procedure completion, i. e. return value GASPI_SUCCESS, the allocated GASPI specific resources of the invoking GASPI process have been released. That means in particular that the communication infrastructure is shut down, all committed groups are released and all allocated segments are freed.

In case of timeout, i. e. return value GASPI_TIMEOUT, the local resources of the invoking GASPI process could not be completely released in the given period of time. A subsequent invocation is required to completely release all of the resources.

In case of error, i. e. return value GASPI_ERROR, the resources of the local GASPI process could not be released. The process is in an undefined state.

5.3.5 gaspi_proc_kill

gaspi_proc_kill sends an interrupt signal to a given GASPI process. It is a *synchronous non-local time-based blocking* procedure.

```
GASPI_PROC_KILL ( rank
                , timeout )
```

Parameter:

(in) rank: the rank of the process to be killed

(in) timeout: the timeout

```
gaspi_return_t
gaspi_proc_kill ( gaspi_rank_t rank
                , gaspi_timeout_t timeout )
```

```
function gaspi_proc_kill(rank,timeout_ms) &
&         result( res ) bind(C, name="gaspi_proc_kill")
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_proc_kill
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

gaspi_proc_kill sends an interrupt signal to the GASPI process incorporating the rank given by parameter *rank*. This can be used, for example, to realise the registration of a user defined signal handler function which ensures the controlled shut down of an entire GASPI application at the global level if the application receives an interrupt signal (*STRG + C*) in the interactive master process. Every GASPI application should register such or a similar signal handler (c. f. listing 9).

In case of successful procedure completion, i. e. return value `GASPI_SUCCESS`, the remote GASPI process has been terminated.

In case of timeout, i. e. return value `GASPI_TIMEOUT`, the remote GASPI process could not be terminated in the given time. A subsequent invocation of the procedure is needed in order to complete the operation.

In case of error, i. e. return value `GASPI_ERROR`, the state of the remote GASPI process is undefined.

> *User advice:* The kill signal terminates a GASPI process in an uncontrolled way. In this case, in order to provide a clean shutdown, it is advisable to register a user defined signal callback function which guarantees a clean shutdown. ⌟

5.3.6 Example

The listing 7 shows a GASPI "Hello world" example. Please note that this example does not deal with failures.

Listing 7: GASPI hello world example.

```
#include <stdio.h>
#include <stdlib.h>
#include <GASPI.h>

int
main (int argc, char *argv[])
{
  gaspi_proc_init (GASPI_BLOCK);

  gaspi_rank_t iProc;
  gaspi_rank_t nProc;

  gaspi_proc_rank (&iProc);
  gaspi_proc_num (&nProc);

  printf ("Hello world from rank %i of %i!\n", iProc, nProc);

  gaspi_proc_term (GASPI_BLOCK);

  return EXIT_SUCCESS;
}
```

Correspondingly the fortran version of GASPI "Hello world" assumes the form listing 8

Listing 8: GASPI hello world example in f90.

```
program hello_world

  use gaspi_c_binding
  implicit none
  integer(gaspi_return_t) :: res
  integer(gaspi_rank_t) :: rank, num
  integer(gaspi_timeout_t) :: timeout

  timeout = GASPI_BLOCK

  res = gaspi_proc_init(timeout)
  res = gaspi_proc_rank(rank)
  res = gaspi_proc_num(num)

  print *,"Hello world from rank ",rank

  res = gaspi_proc_term(timeout)

end program hello_world
```

The listing 9 shows the registration of a user defined signal handler function which ensures the controlled shut down of an entire GASPI application at the global level if the application receives an interrupt signal (*STRG + C*) in the interactive master process. Every GASPI application should register such or a similar signal handler.

Listing 9: Signal handling.

```
#include <signal.h>
#include <stdlib.h>
#include <GASPI.h>

void
signalHandler (int sigint)
{
  gaspi_rank_t iProc;
  gaspi_rank_t nProc;

  gaspi_proc_rank (&iProc);
  gaspi_proc_num (&nProc);

  if (0 == iProc)
    {
      for (iProc = 1; iProc < nProc; ++iProc)
        {
          gaspi_proc_kill (iProc, GASPI_BLOCK);
        }
    }

```

```
  gaspi_proc_term (GASPI_BLOCK);

  exit (EXIT_FAILURE);
}


int
main (int argc, char *argv[])
{
  gaspi_proc_init (GASPI_BLOCK);

  signal (SIGINT, &signalHandler);

  /* working phase */

  gaspi_proc_term (GASPI_BLOCK);

  return EXIT_SUCCESS;
}
```

5.4 Connection management utilities

5.4.1 gaspi_connect

In order to be able to communicate between two GASPI processes, the communication infrastructure has to be established. This is achieved with the *synchronous non-local time-based blocking* procedure `gaspi_connect`. It is bound to the working phase of the GASPI life cycle.

```
GASPI_CONNECT ( rank
              , timeout )
```

Parameter:

(in) rank: the remote rank with which the communication infrastructure is established

(in) timeout: The timeout for the operation

```
gaspi_return_t
gaspi_connect ( gaspi_rank_t rank
              , gaspi_timeout_t timeout )
```

```
function gaspi_connect(rank,timeout_ms) &
&         result( res ) bind(C, name="gaspi_connect")
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

gaspi_connect builds up the communication infrastructure, passive as well as one-sided and atomic operations, between the local and the remote GASPI process representing rank *rank*. The connection is bi-directional, i. e. it is sufficient that gaspi_connect is invoked by only one of the connection partners.

In case of successful procedure completion, i. e. return value GASPI_SUCCESS, the communication infrastructure is established. If there is an allocated segment, the segment can be used as a destination for passive communication between the two nodes. In case the connection has already been established, e. g. by the connection partner, the return value is GASPI_SUCCESS.

In case of return value GASPI_TIMEOUT, the communication infrastructure could not be established between the local GASPI process and the remote GASPI process in the given period of time.

In case of return value GASPI_ERROR, the communication infrastructure could not be established between the local GASPI process and the remote GASPI process.

In case of the latter two return values, a check of the state vector by invocation of gaspi_state_vec_get gives information on whether the remote GASPI process is still healthy.

> *User advice:* Under the assumption that the GASPI process is initialized with parameter *build_infrastructure* set to *true*, all the connections are set up at initialization time. Hence, a subsequent call to gaspi_connect is superfluous in this case. ⌟

5.4.2 gaspi_disconnect

The gaspi_disconnect procedure is a *synchronous local blocking* procedure which disconnects a given process, identified by its rank, and frees all associated resources.

It is bound to the working phase of the GASPI life cycle.

```
GASPI_DISCONNECT ( rank
                 , timeout )
```

Parameter:

(in) rank: the remote rank from which the communication infrastructure is disconnected

(in) timeout: The timeout for the operation

```
gaspi_return_t
gaspi_disconnect ( gaspi_rank_t rank
                 , gaspi_timeout_t timeout )
```

```
function gaspi_disconnect(rank,timeout_ms) &
&         result( res ) bind(C, name="gaspi_disconnect")
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_disconnect
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error

`gaspi_disconnect` disconnects the communication infrastructure, passive as well as one-sided and atomic operations, between the local and the remote GASPI process representing rank *rank*. The connection is bi-directional, i. e. it is sufficient if `gaspi_disconnect` is invoked by only one of the connection partners.

In case of successful procedure completion, i. e. return value `GASPI_SUCCESS`, the communication infrastructure is disconnected. Associated resources are freed on the local as well as on the remote side. In case the connection has already been disconnected, e. g. by the connection partner, the return value is `GASPI_SUCCESS`.

In case of error the return value is `GASPI_ERROR`.

In case of return value `GASPI_TIMEOUT`, the connection between the local GASPI process and the remote GASPI process could not be disconnected in the given period of time.

In case of the latter two return values local resources are freed and a check of the state vector by invocation of `gaspi_state_vec_get` gives information whether the remote GASPI process is still healthy.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the connection is disconnected and can no longer be used.

5.5 State vector for individual processes

5.5.1 Introduction

A necessary pre-condition for realising a failure tolerant code is a detailed knowledge about the state of the communication partners of each local GASPI process.

GASPI provides a predefined type to describe the state of a remote GASPI process, which is the `gaspi_state_t` type. `gaspi_state_t` can have one of two values:

GASPI_STATE_HEALTHY implies that the remote GASPI process is healthy, i. e. communication is possible.

GASPI_STATE_CORRUPT means that the remote GASPI process is corrupted, i. e. there is no communication possible.

```
typedef vector<gaspi_state_t> gaspi_state_vector_t
```

`gaspi_state_vector_t` *is a vector with state information for individual processes. The length of the vector equals the number of processes in the* GASPI *program and the entries are ordered based on the process ranks, i. e. entry 0 of the vector represents the state of process with the rank 0.* ⌟

There are procedures to query the state of the communication partners after a given communication request and also to reset the state after successful recovery. These are described in the following subsections.

The state vector does not provide a global view, instead each process has its own state vector that may be different to the state vector of another process.

5.5.2 gaspi_state_vec_get

The state vector is obtained by the *local synchronous blocking* function `gaspi_state_vec_get`.

The state vector represents the states of all GASPI processes.

```
GASPI_STATE_VEC_GET ( state_vector )
```

Parameter:

(out) returns: the vector with individual return codes

```
gaspi_return_t
gaspi_state_vec_get ( gaspi_state_vector_t *state_vector )
```

```
function gaspi_state_vec_get(state_vector) &
&         result( res ) bind(C, name="gaspi_state_vec_get")
  type(c_ptr), value :: state_vector
  integer(gaspi_return_t) :: res
end function gaspi_state_vec_get
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

The state vector has one entry for each rank. It is created and initialized during gaspi_proc_init. It is updated, case required, in each of the following operations:

- group commitment
  - gaspi_group_commit
- segment registration
  - gaspi_segment_register
- one-sided communication
  - gaspi_wait
  - gaspi_write
  - gaspi_read
  - gaspi_write_list
  - gaspi_read_list
  - gaspi_notify
  - gaspi_write_notify
  - gaspi_write_list_notify
- passive communication
  - gaspi_passive_send
  - gaspi_passive_receive
- collective operations
  - gaspi_barrier
  - gaspi_allreduce
  - gaspi_allreduce_user
- global atomic operations
  - gaspi_atomic_fetch_and_add
  - gaspi_atomic_compare_swap

An update is not guaranteed to update all entries in the state vector, but may only update the entries of the direct communication partners. `gaspi_state_vec_get` retrieves in case of successful completion, i. e. return value `GASPI_SUCCESS`, the state vector. It contains the states of the GASPI processes with which the local process has been communicating. All other entries are unmodified.

In case of error, the return value is `GASPI_ERROR` and the value of the state vector is undefined.

> *User advice:* For failure tolerant code, the state vector should be checked after each of the above procedure calls in case they return with either return value `GASPI_ERROR` or `GASPI_TIMEOUT`. ⌟

5.6 MPI Interoperability

GASPI aims at providing interoperability with MPI in order to allow for incremental porting of such applications.

The startup of mixed MPI and GASPI code is achieved by invoking `gaspi_proc_init` in an existing MPI program. This way, MPI takes care of distributing and starting the binary and GASPI just takes care of setting up its internal infrastructure.

GASPI and MPI communication should not occur at the same time, i. e. only the program layout given in Listing 10 is supported

Listing 10: Embedded GASPI program

```
mpi_startup;

/* MPI part, no ongoing GASPI communication... */

/* ...finish all ongoing MPI communication */

mpi_barrier;

/* no ongoing MPI communication */

gaspi_proc_init;

while (!done) {

  /* GASPI part, no ongoing MPI communication... */

  /* ...finish all ongoing GASPI communication */

  gaspi_barrier;

  /* MPI part, no ongoing GASPI communication... */

  /* ...finish all ongoing MPI communication */
```

```

  mpi_barrier;
}

gaspi_proc_term;

/* MPI part, no ongoing GASPI communication */

mpi_shutdown;
```

5.7 Argument checks and performance

GASPI aims at high performance and does not provide any argument checks at procedure invocation per default.

> *Implementor advice:* The implementation should provide a specific library which includes argument checks. The GASPI procedures should include out of bounds checks, there. ⌟

6 Groups

6.1 Introduction

Groups are subsets of the total number of GASPI processes. The group members have common collective operations. Each GASPI process may participate in more than one group.

The use-cases are the collective operations provided in section 11 that make sense to be performed only for a subset of GASPI processes in order to avoid a complete (all processes) collective synchronisation point.

A group has to be defined and declared in each of the participating GASPI processes. Defining a group is a three step procedure. An empty group has to be created first. Then the participating GASPI processes, represented by their ranks, have to be attached. The group definition is a local operation. In order to activate the group, the group has to be committed by each of the participating GASPI processes. This is a collective operation for the group. Only after a successful group commit, can the group be used for collective operations.

The maximum number of groups allowed per GASPI process is restricted by the implementation. A user defined value can be set with `gaspi_config_set` before initialization (`gaspi_proc_init`).

In case one group desintegrates due to some failure, the group has to be re-established. If there is a new process replacing the failed one, the group has to be defined and declared on the newly started GASPI process(es). Re-establishment of the group is then achieved by recommitment of the group by the GASPI processes which were still 'alive' (functioning) and by the newly started GASPI process.

6.2 GASPI group generics

6.2.1 GASPI group type

Groups are specified with a special group type `gaspi_group_t`.

6.2.2 GASPI_GROUP_ALL

`GASPI_GROUP_ALL` is a predefined default group that corresponds to the whole set of GASPI processes. This is to be used for collective operations that work for the whole system.

```
gaspi_group_t GASPI_GROUP_ALL;
```

> *User advice:* Note that `GASPI_GROUP_ALL` is a group definition like any other sub group. In order to be used, `GASPI_GROUP_ALL` also has to be committed by `gaspi_group_commit`. ⌟

6.3 Group creation

6.3.1 gaspi_group_create

The `gaspi_group_create` procedure is a *synchronous local blocking* procedure which creates an empty group.

```
GASPI_GROUP_CREATE ( group )
```

Parameter:

(out) group: the created empty group

```
gaspi_return_t
gaspi_group_create ( gaspi_group_t *group )
```

```
function gaspi_group_create(group) &
&         result( res ) bind(C, name="gaspi_group_create")
  integer(gaspi_group_t) :: group
  integer(gaspi_return_t) :: res
end function gaspi_group_create
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, *group* represents an empty group without any members.

In case of error, the return value is GASPI_ERROR.

6.3.2 gaspi_group_add

The gaspi_group_add procedure is a *synchronous local blocking* procedure which adds a given rank to an existing group.

```
GASPI_GROUP_ADD ( group
                , rank )
```

Parameter:

(inout) group: the group to which the rank is added

(in) rank: the rank to add to the group

```
gaspi_return_t
gaspi_group_add ( gaspi_group_t group
                , gaspi_rank_t rank )
```

```
function gaspi_group_add(group,rank) &
&         result( res ) bind(C, name="gaspi_group_add")
  integer(gaspi_group_t), value :: group
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_return_t) :: res
end function gaspi_group_add
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, the GASPI process with *rank* is added to *group*. Whenever you add a rank the list of ranks is sorted in ascending order.

In case of error, the return value is GASPI_ERROR.

6.3.3 gaspi_group_commit

The gaspi_group_commit procedure is a *synchronous collective time-based blocking* procedure which establishes a group.

```
GASPI_GROUP_COMMIT ( group
                   , timeout )
```

Parameter:

(in) group: the group to commit

(in) timeout: the timeout

```
gaspi_return_t
gaspi_group_commit ( gaspi_group_t group
                   , gaspi_timeout_t timeout )
```

```
function gaspi_group_commit(group,timeout_ms) &
&         result( res ) bind(C, name="gaspi_group_commit")
  integer(gaspi_group_t), value :: group
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_group_commit
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

The group committed by all participating processes must contain all ranks and must identical for all processes, otherwise the result is undefined.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the group given by the parameter *group* is established. Collective operations invoked by the members of the group are allowed from this moment on.

In case of timeout, i. e. return value GASPI_TIMEOUT, the group could not be established on all ranks forming the group in the given period of time. The group is in an undefined state and collective operations on the group yield undefined behavior. A subsequent invocation is required in order to completely establish the group.

In case of error, i. e. return value GASPI_ERROR, the group could not be established. The group is in an undefined state and collective operations defined on the given group yield undefined behavior.

In both cases, GASPI_TIMEOUT and GASPI_ERROR, the GASPI state vector should be checked in order to eliminate the possibility of a failure.

> *User advice:* Any group commit should be performed only by a single thread of a process. If two GASPI processes are members of two groups, then the order of the group commits should be the same on both processes in order to avoid deadlocks. ⌟

> *Implementor advice:* If the parameter *build_infrastructure* is not set, the procedure gaspi_group_commit must set up the infrastructure for all possible operations of the group. ⌟

6.4 Group deletion

6.4.1 gaspi_group_delete

The gaspi_group_delete procedure is a *synchronous local blocking* procedure which deletes a given group.

```
GASPI_GROUP_DELETE ( group )
```

Parameter:

(in) group: the group to be deleted

```
gaspi_return_t
gaspi_group_delete ( gaspi_group_t group )
```

```
function gaspi_group_delete(group) &
&         result( res ) bind(C, name="gaspi_group_delete")
  integer(gaspi_group_t), value :: group
  integer(gaspi_return_t) :: res
end function gaspi_group_delete
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, *group* is deleted and cannot be used further.

In case of error, the return value is GASPI_ERROR.

> *Implementor advice:* If the parameter *build_infrastructure* is not set to true, the procedure gaspi_group_delete must disconnect all connections which have been set up in the call to gaspi_group_commit and free all associated resources. ⌟

6.5 Group utilities

6.5.1 gaspi_group_num

The gaspi_group_num procedure is a *synchronous local blocking* procedure which returns the current number of allocated groups.

```
GASPI_GROUP_NUM ( group_num )
```

Parameter:

(out) group_num: the current number of groups

```
gaspi_return_t
gaspi_group_num ( gaspi_number_t *group_num )
```

```
function gaspi_group_num(group_num) &
&         result( res ) bind(C, name="gaspi_group_num")
  integer(gaspi_number_t) :: group_num
  integer(gaspi_return_t) :: res
end function gaspi_group_num
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

After successful procedure completion, i. e. return value GASPI_SUCCESS, *group_num* contains the current number of allocated groups. The value of *group_num* is related to the parameter *group_max* in the configuration structure and cannot exceed that value. The value can be implementation specific.

6.5.2 gaspi_group_size

The gaspi_group_size procedure is a *synchronous local blocking* procedure which returns the number of ranks of a given group.

```
GASPI_GROUP_SIZE ( group
                 , group_size )
```

Parameter:

(in) group: the group to be examined

(out) group_size: the number of ranks in a given group

```
gaspi_return_t
gaspi_group_size ( gaspi_group_t group
                 , gaspi_number_t *group_size )
```

```
function gaspi_group_size(group,group_size) &
&         result( res ) bind(C, name="gaspi_group_size")
  integer(gaspi_group_t), value :: group
  integer(gaspi_number_t) :: group_size
  integer(gaspi_return_t) :: res
end function gaspi_group_size
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

After successful procedure completion, i. e. return value GASPI_SUCCESS, *group_size* contains the number of GASPI processes forming the *group*.

In case of error, the return value is GASPI_ERROR. The parameter *group_size* has an undefined value.

6.5.3 gaspi_group_ranks

The gaspi_group_ranks procedure is a *synchronous local blocking* procedure which returns a list of ranks of GASPI processes forming the group.

```
GASPI_GROUP_RANKS ( group
                  , group_ranks[group_size] )
```

Parameter:

(in) group: the group to be examined

(out) group_ranks: the list of ranks forming the group

```
gaspi_return_t
gaspi_group_ranks ( gaspi_group_t group
                  , gaspi_rank_t *group_ranks )
```

```
function gaspi_group_ranks(group,group_ranks) &
&         result( res ) bind(C, name="gaspi_group_ranks")
  integer(gaspi_group_t), value :: group
  type(c_ptr), value :: group_ranks
  integer(gaspi_return_t) :: res
end function gaspi_group_ranks
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

After successful procedure completion, i. e. return value GASPI_SUCCESS, the list *group_ranks* contains the ranks of the processes that belong to the *group*. The list is not allocated by the procedure. The list allocation is supposed to be done outside of the procedure. The size of the list can be inquired by gaspi_group_size.

In case of error, the return value is GASPI_ERROR. The list *group_ranks* has an undefined value.

7 GASPI segments

7.1 Introduction and overview

Modern hardware has a complex memory hierarchy with different bandwidth and latencies for read and write accesses. Among them are non-uniform memory access (*NUMA*) partitions, solid state devices (*SSD*s), graphical processing unit (*GPU*) memory or many integrated cores (*MIC*) memory.

The GASPI memory segments are thus an abstraction representing any kind of memory level, mapping the variety of hardware layers to the software layer. A segment is a contiguous block of virtual memory. In the spirit of the PGAS approach, these GASPI segments may be globally accessible from every thread of every GASPI process and represent the partitions of the global address space.

By means of the GASPI memory segments it is also possible for multiple memory models or indeed multiple applications to share a single Partitioned Global Address Space.

Since segment allocation is expensive and the total number of supported segments is limited due to hardware constraints, the GASPI memory management paradigm is the following. GASPI provides only a few relatively large segments. Allocations inside of the pre-allocated segment memory are managed by the application.

Every GASPI process may possess a certain number of segments (not necessarily equal to the number possessed by the other ranks) that may be accessed as common memory, whether locally —with normal memory operations—or remotely—with the communication routines of GASPI.

In order to use a segment for communication between two processes, some setup steps are required in general.

A memory segment has to be allocated in each of the processes by the *local* procedure `gaspi_segment_alloc`. In order to also use the segments for one-sided communication, the memory segment has to be registered on the remote process which will access the memory segment at some point. This is achieved by the *non-local* procedure `gaspi_segment_register`.

> *User advice:* If the parameter *build_infrastructure* is not set, a connection has to be established between the processes before the segment can be registered at the remote process. This is accomplished by calling the procedure `gaspi_connect`. ⌟

`gaspi_segment_create` unites these steps into a single *collective* procedure for an entire group. After successful procedure completion, a common segment is created on each GASPI process forming the group which can be immediately used for communication among the group members.

During the lifetime of an application no segment is available unless it is explicitly created with `gaspi_segment_alloc` or `gaspi_segment_create` after the GASPI startup.

7.2 Segment creation

7.2.1 `gaspi_segment_alloc`

The *synchronous local blocking* procedure `gaspi_segment_alloc` allocates a memory segment and optionally maps it in accordance with a given allocation policy.

```
GASPI_SEGMENT_ALLOC ( segment_id
                    , size
                    , alloc_policy )
```

Parameter:

(in) segment_id: The segment ID to be created. The segment IDs need to be unique on each GASPI process

(in) size: The size of the segment in bytes

(in) alloc_policy: allocation policy

```
gaspi_return_t
gaspi_segment_alloc ( gaspi_segment_id_t segment_id
                    , gaspi_size_t size
                    , gaspi_alloc_t alloc_policy )
```

```
function gaspi_segment_alloc(segment_id,size,alloc_policy) &
&         result( res ) bind(C, name="gaspi_segment_alloc")
  integer(gaspi_segment_id_t), value :: segment_id
  integer(gaspi_size_t), value :: size
  integer(gaspi_alloc_t), value :: alloc_policy
  integer(gaspi_return_t) :: res
end function gaspi_segment_alloc
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_segment_alloc` allocates a segment of size *size* that will be referenced by the *segment_id* identifier. This identifier parameter has to be unique in the local GASPI process. Creating a new segment with an existing segment ID results in undefined behavior. Note that the total number of segments is restricted by the underlying hardware capabilities. The maximum number of supported segments can be retrieved by invoking `gaspi_segment_max`.

Allocation of segments in GASPI allows for various so-called policies. The default policy in a cc-numa mode for example might be an allocation of socket-local memory, a different policy might allow to map GPU memory into the main memory of the host and yet another policy might allow for a direct access of external non-volatile RAM.

The *alloc_policy* is used to pass an allocation policy. The default allocation policy behavior is left to the implementation. The default allocation parameter is GASPI_ALLOC_DEFAULT.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the segment can be accessed locally. In case that there is a connection established to a remote GASPI process, it can also be used for passive communication between the two GASPI processes. (Note that this is always the case if the process has been initialized with the parameter *build_infrastructure* set to *true*), it can also be used for passive communication between the two GASPI processes; either as a source segment for `gaspi_passive_send` or as a destination segment for `gaspi_passive_receive`.

A return value `GASPI_ERROR` indicates that the segment allocation failed and the segment cannot be used.

> *User advice:* A GASPI implementation may allocate more memory than requested by the application for internal management. ⌟

Implementor advice: In case of non-uniform memory access architectures, the memory should be allocated close to the calling process. The allocation policy of the calling process should not be modified. ⌟

7.2.2 gaspi_segment_register

In order to be used in a one-sided communication request on an existing connection, a segment allocated by `gaspi_segment_alloc` needs to be made visible and accessible for the other GASPI processes. This is accomplished by the procedure `gaspi_segment_register`. It is a *synchronous non-local time-based blocking* procedure.

```
GASPI_SEGMENT_REGISTER ( segment_id
                       , rank
                       , timeout )
```

Parameter:

(in) segment_id: The segment ID to be registered. The segment ID's need to be unique for each GASPI process

(in) rank: The rank of the GASPI process which should register the new segment

(in) timeout: The timeout for the operation

```
gaspi_return_t
gaspi_segment_register ( gaspi_segment_id_t segment_id
                       , gaspi_rank_t rank
                       , gaspi_timeout_t timeout )
```

```
function gaspi_segment_register(segment_id,rank,timeout_ms) &
&         result( res ) bind(C, name="gaspi_segment_register")
  integer(gaspi_segment_id_t), value :: segment_id
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_segment_register
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_segment_register` makes the segment referenced by the *segment_id* identifier visible and accessible to the GASPI process with the associated *rank*.

> *User advice:* If the parameter *build_infrastructure* is not set, a connection has to be established between the processes before the segment can be registered at the remote process. This is accomplished calling the procedure `gaspi_connect`. ⌟

In case of successful procedure completion, i. e. return value `GASPI_SUCCESS`, the local segment can be used for one-sided communication requests which are invoked by the given remote process.

In case of return value `GASPI_TIMEOUT`, the segment could not be registered in the given period of time. The segment cannot be used for one-sided communication requests which are invoked by the given remote process. A subsequent call of `gaspi_segment_register` has to be invoked in order to complete the registration request.

In case of return value `GASPI_ERROR`, the segment could not be registered on the remote side. The segment cannot be used for one-sided communication requests which are invoked by the given remote process.

In case of the latter two return values, a check of the state vector by invocation of `gaspi_state_vec_get` gives information as to whether or not the remote GASPI process is still healthy.

> *User advice:* Note that a local return value `GASPI_SUCCESS` does not imply that the remote process is informed explicitly that the segment is accessible. This can be achieved through an explicit synchronisation, either by one of the collective operations or by an explicit notification. ⌟

7.2.3 `gaspi_segment_create`

`gaspi_segment_create` is a *synchronous collective time-based blocking* procedure. It is semantically equivalent to a collective aggregation of `gaspi_segment_alloc`, `gaspi_segment_register` and `gaspi_barrier` involving all of the members of a given group. If the communication infrastructure was not established for all group members beforehand, `gaspi_segment_create` will accomplish this as well.

```
GASPI_SEGMENT_CREATE ( segment_id
                     , size
                     , group
                     , timeout
                     , alloc_policy )
```

Parameter:

(in) segment_id: The ID for the segment to be created. The segment ID's need to be unique for each GASPI process

(in) size: The size of the segment in bytes

(in) group: The group which should create the segment

(in) timeout: The timeout for the operation

(in) alloc_policy: allocation policy

```
gaspi_return_t
gaspi_segment_create ( gaspi_segment_id_t segment_id
                     , gaspi_size_t size
                     , gaspi_group_t group
                     , gaspi_timeout_t timeout
                     , gaspi_alloc_t alloc_policy )
```

```
function gaspi_segment_create(segment_id,size,group, &
&         timeout_ms,alloc_policy) &
&         result( res ) bind(C, name="gaspi_segment_create")
  integer(gaspi_segment_id_t), value :: segment_id
  integer(gaspi_size_t), value :: size
  integer(gaspi_group_t), value :: group
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_alloc_t), value :: alloc_policy
  integer(gaspi_return_t) :: res
end function gaspi_segment_create
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_segment_create` allocates a segment of size *size* that will be referenced by the *segment_id* identifier. This identifier parameter has to be unique on the local GASPI process. Creating a new segment with an existing segment ID results in undefined behavior. `gaspi_segment_create` makes the segment referenced by the *segment_id* identifier visible and accessible to all of the GASPI processes forming the group *group*. The maximum number of supported segments can be retrieved by invoking `gaspi_segment_max`. The *alloc_policy* is used to pass an allocation policy. The default allocation policy behavior is left to the implementation.

After successful procedure completion, i. e. `GASPI_SUCCESS`, the segment can be accessed locally and it can be used as a destination for the passive communication channel. Either as a source segment for `gaspi_passive_send` or as a destination segment for `gaspi_passive_receive`. Furthermore, it can be used for one-sided communication requests, which are invoked by the remote processes forming the group *group* or global atomic operations. The segment *segment_id* is ready to be used.

For consistency and programs with hard failure tolerance requirements, the operation must be performed within *timeout* milliseconds. In case of return value `GASPI_TIMEOUT`, progress has been achieved, however the operation could not be completed in the given timeout. The segment cannot be used locally neither remotely. The segment cannot be used for one-sided or passive communication requests which are invoked by the other remote processes forming the group. The same applies to global atomic operations. A subsequent call of `gaspi_segment_create` has to be invoked in order to complete the segment creation.

In case of return value `GASPI_ERROR`, the segment creation failed in one of the above progress steps on at least one of the involved GASPI processes. The segment cannot be used locally neither remotely. The segment cannot be used for one-sided or passive communication requests which are invoked by the other remote processes forming the group. The same applies to global atomic operations.

In case of the latter two return values, a check of the state vector by invocation of `gaspi_state_vec_get` gives information whether the involved remote GASPI processes are still healthy.

> *User advice:* A GASPI implementation may allocate more memory than requested by the application for internal management. ⌟

> *Implementor advice:* In case of non-uniform memory access architectures, the memory should be allocated close to the calling process. The allocation policy of the calling process should not be modified. ⌟

7.2.4 `gaspi_segment_bind`

The *synchronous local blocking* procedure `gaspi_segment_bind` binds a segment id to user provided memory.

```
GASPI_SEGMENT_BIND ( segment_id
                   , pointer
                   , size
                   , memory_description
                   )
```

Parameter:

(in) segment_id: Unique segment ID to bind.

(in) pointer: The begin of the memory provided by the user.

(in) size: The size of the memory provided by *pointer* in bytes.

(in) memory_description: The description of the memory provided.

```
gaspi_return_t gaspi_segment_bind
  ( gaspi_segment_id_t const segment_id
  , gaspi_pointer_t const pointer
  , gaspi_size_t const size
  , gaspi_memory_description_t const memory_description
  )
```

```
function gaspi_segment_bind ( segment_id                  &
&                           , pointer                     &
&                           , size                        &
&                           , memory_description          &
&                           )                             &
&        result (res) bind (C, name="gaspi_segment_bind")
  integer (gaspi_segment_id_t), value :: segment_id
  type (c_ptr), value :: pointer
  integer (gaspi_size_t), value :: size
  integer (gaspi_memory_description_t), value :: memory_description
  integer (gaspi_return_t) :: res
end function gaspi_segment_bind
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_segment_bind` binds the segment identified by the identifier *segment_id* to the user provided memory of size *size* located at the address *pointer*. To provide less than *size* bytes results in undefined behavior. The identifier *segment_id* must be unique in the local GASPI process. Bind to a segment with an existing segment ID (regardless of bind or allocated) results in undefined behavior. Note that the total number of segments is restricted by the underlying hardware capabilities. The maximum number of supported segments can be retrieved by invoking `gaspi_segment_max`.

To bind successfully the user provided memory must satisfy implementation specific constraints, e. g. alignment constraints.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the segment can be accessed locally and has the same capabilities like a segment that was allocated by a successful call to `gaspi_segment_alloc`.

If the procedure returns with `GASPI_ERROR`, the bind has failed and the segment can not be used.

> *User advice:* A GASPI implementation may allocate additional memory for internal management. Depending on the implementation it might be required that the management memory must reside on the same device as the provided memory. ⌟

7.2.5 gaspi_segment_use

The *synchronous collective time-based blocking* procedure `gaspi_segment_use` is semantically equivalent to a collective aggregation of `gaspi_segment_bind`, `gaspi_segment_register` and `gaspi_barrier` involving all members of a given group. If the communication infrastructure was not established for all group members beforehand, `gaspi_segment_use` will accomplish this as well.

```
GASPI_SEGMENT_USE ( segment_id
                  , pointer
                  , size
                  , group
                  , timeout
                  , memory_description
                  )
```

Parameter:

(in) segment_id: Unique segment ID to bind.

(in) pointer: The begin of the memory provided by the user.

(in) size: The size of the memory provided by *pointer* in bytes.

(in) group: The group which should create the segment.

(in) timeout: The timeout for the operation.

(in) memory_description: The description of the memory provided.

```
gaspi_return_t gaspi_segment_use
  ( gaspi_segment_id_t const segment_id
  , gaspi_pointer_t const pointer
  , gaspi_size_t const size
  , gaspi_group_t const group
  , gaspi_timeout_t const timeout
  , gaspi_memory_description_t const memory_description
  )
```

```
function gaspi_segment_use ( segment_id             &
&                          , pointer                &
&                          , size                   &
&                          , group                  &
&                          , timeout                &
&                          , memory_description     &
&                          )                        &
&        result (res) bind (C, name="gaspi_segment_use")
  integer (gaspi_segment_id_t), value :: segment_id
  type (c_ptr), value :: pointer
  integer (gaspi_size_t), value :: size
  integer (gaspi_group_t), value :: group
  integer (gaspi_timeout_t), value :: timeout
  integer (gaspi_memory_description_t), value :: memory_description
  integer (gaspi_return_t) :: res
end function gaspi_segment_bind
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

gaspi_segment_use binds the segment identified by the identifier *segment_id* to the user provided memory of size *size* located at the address *pointer*. To provide a *size* larger than the actual buffer size pointed by *pointer* results in undefined behavior. gaspi_segment_use makes the segment referenced by the *segment_id* identifier visible and accessible to all of the GASPI processes forming the group *group*. The identifier *segment_id* must be unique in the local GASPI process. Attempting to use an existing segment ID (regardless of bind or allocated) results in undefined behavior. Note that the total number of segments is restricted by the underlying hardware capabilities. The maximum number of supported segments can be retrieved by invoking gaspi_segment_max.

To use successfully the user provided memory must satisfy implementation specific constraints, e. g. alignment constraints.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the segment can be accessed globally and has the same capabilities like a segment that was created by a successful call to gaspi_segment_create.

In case of return value GASPI_TIMEOUT the operation could not be completed in the given timeout. The segment cannot be used locally neither remotely. A subsequent call of gaspi_segment_use has to be invoked in order to complete the request.

If the procedure returns with GASPI_ERROR, the procedure has failed and the segment can not be used.

Implementor advice: gaspi_segment_use can be formulated in pseudo code as

```
GASPI_SEGMENT_USE (id, pointer, size, group, timeout, memory)
{
  GASPI_SEGMENT_BIND (id, pointer, size, memory);

  foreach (rank : group)
  {
    timeout -= GASPI_CONNECT (rank, timeout);
    timeout -= GASPI_SEGMENT_REGISTER (id, rank, timeout);
  }

  GASPI_BARRIER (group, timeout);
}
```

where the call gets executed on all members of *group*.

7.3 Segment deletion

7.3.1 gaspi_segment_delete

The *synchronous local blocking* procedure gaspi_segment_delete releases the resources of a previously allocated memory segment.

```
GASPI_SEGMENT_DELETE ( segment_id )
```

Parameter:

(in) segment_id: The segment ID to be deleted.

```
gaspi_return_t
gaspi_segment_delete ( gaspi_segment_id_t segment_id )
```

```
function gaspi_segment_delete(segment_id) &
&         result( res ) bind(C, name="gaspi_segment_delete")
  integer(gaspi_segment_id_t), value :: segment_id
  integer(gaspi_return_t) :: res
end function gaspi_segment_delete
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error `gaspi_segment_delete` releases the resources of the segment which is referenced by the *segment_id* identifier.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the segment is deleted and the resources are released. It would be an application error to use the segment for communication between two GASPI processes after `gaspi_delete` has been called.

In case of return value `GASPI_ERROR`, the segment deletion failed. The segment is in an undefined state and cannot be used locally neither remotely. The segment cannot be used for one-sided or passive communication requests which are invoked by the other remote processes forming the group. The same applies to global atomic operations.

7.4 Segment utilities

7.4.1 gaspi_segment_num

The `gaspi_segment_num` procedure is a *synchronous local blocking* procedure which returns the current number of allocated segments.

```
GASPI_SEGMENT_NUM ( segment_num )
```

Parameter:

(out) segment_num: the current number of allocated segments

```
gaspi_return_t
gaspi_segment_num ( gaspi_number_t *segment_num )
```

```
function gaspi_segment_num(segment_num) &
&         result( res ) bind(C, name="gaspi_segment_num")
  integer(gaspi_number_t) :: segment_num
  integer(gaspi_return_t) :: res
end function gaspi_segment_num
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, *segment_num* contains the current number of locally allocated segments provided by GASPI. The value of *segment_num* is related to the parameter *segment_max* in the configuration structure which is retrieved by `gaspi_config_get` and cannot exceed that value. The maximum number of allocatable segments per process might be implementation specific.

In case of error, the return value is `GASPI_ERROR`. The parameter *segment_num* has an undefined value.

7.4.2 `gaspi_segment_list`

The `gaspi_segment_list` procedure is a *synchronous local blocking* procedure which returns a list of locally allocated segment IDs.

```
GASPI_SEGMENT_LIST ( num
                   , segment_id_list[num] )
```

Parameter:

(in) num: number of segment IDs to collect

(out) segment_list[num]: list of locally allocated segment IDs

```
gaspi_return_t
gaspi_segment_list ( gaspi_number_t num
                   , gaspi_segment_id_t *segment_id_list )
```

```
function gaspi_segment_list(num,segment_id_list) &
&         result( res ) bind(C, name="gaspi_segment_list")
  integer(gaspi_number_t), value :: num
  type(c_ptr), value :: segment_id_list
  integer(gaspi_return_t) :: res
end function gaspi_segment_list
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, *segment_id_list[num]* contains the IDs of *num* locally allocated segments.

The size of *segment_id_list[num]* needs to be at least *num* elements long.

In case of error, the return value is `GASPI_ERROR`. The parameter *segment_list[num]* has an undefined value.

7.4.3 gaspi_segment_ptr

Segments are identified by a unique ID. This ID can be used to obtain the virtual address of that local segment of memory. The procedure `gaspi_segment_ptr` returns the pointer to the segment represented by a given segment ID. It is a *synchronous local blocking* procedure.

```
GASPI_SEGMENT_PTR ( segment_id
                  , pointer )
```

Parameter:

(in) segment_id: The segment ID.

(out) pointer: The pointer to the memory segment.

```
gaspi_return_t
gaspi_segment_ptr ( gaspi_segment_id_t segment_id
                  , gaspi_pointer_t *pointer )
```

```
function gaspi_segment_ptr(segment_id,ptr) &
&         result( res ) bind(C, name="gaspi_segment_ptr")
  integer(gaspi_segment_id_t), value :: segment_id
  type(c_ptr) :: ptr
  integer(gaspi_return_t) :: res
end function gaspi_segment_ptr
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. `GASPI_SUCCESS`, the output parameter *pointer* contains the virtual address pointer of the memory identified by *segment_id*. This `gaspi_pointer_t` can then be used to reference the segment and perform memory operations.

In case of return value `GASPI_ERROR`, the translation of the segment ID to a pointer to a virtual memory address failed. The pointer contains an undefined value and cannot be used to reference the segment.

7.5 Segment memory management

Each thread of a process may have global read or write access to all of the segments provided by remote GASPI processes if there is a connection established between the processes and if the respective segments have been registered on the local process.

Since a segment is an entire contiguous block of virtual memory, allocations inside of the pre-allocated segment memory need to be managed.

GASPI does not provide dedicated memory management functionality for the local segments. This is left to the application. Since a default implementation for memory management cannot include knowledge about the specific problem, a good problem-related implementation of a memory management will always better than any predefined implementation.

Local and *non-local* GASPI procedures specify in general memory addresses within the Partitioned Global Address Space by the triple consisting of a rank, a segment identifier and an offset. This prevents a global all-to-all distribution of memory addresses, since memory addresses of memory segments could be and normally are different on different GASPI processes.

A local buffer is specified by the pair *segment_id*, *offset*. The buffer is located at address

$$\mathrm{buffer\_address} = \mathrm{base\_addr}\,(\,\mathrm{segment\_id}\,) + \mathrm{offset}$$

where base_addr(segment_id) is the base address of the segment with identifier *segment_id*. It can be obtained by applying `gaspi_segment_ptr` on the local process.

A remote buffer is specified by the triple *remote_rank*, *remote_segment_id*, *remote_offset*. The address of the remote buffer can be calculated analogously to the local buffer. The only difference is the determination of the base address. Here, it is the address which would be obtained by invoking `gaspi_segment_ptr` on the remote GASPI process with *remote_segment_id* as input parameter.

8 One-sided communication

8.1 Introduction and overview

One-sided asynchronous communication is the basic communication mechanism provided by GASPI. Hereby, one GASPI process specifies all communication parameters, both for the local and the remote side. Due to the asynchronicity, a complete communication involves two procedure calls. First, one call to initiate the communication. This call posts a communication request to the underlying network infrastructure. The second call waits for the completion of the communication request.

For one-sided communication, GASPI provides the concept of communication queues. All operations placed on a certain queue q by one or several threads are finished after a single wait call on the queue q has returned successfully. Separation of concerns is possible by using different queues for different tasks, e. g. one queue for operations on data and another queue for operations on meta-data.

The different communication queues guarantee fair communication, i. e. no queue should see its communication requests delayed indefinitely.

One-sided communication calls can basically be divided into two operation types: read and write. The read operations transfer data from a remote segment to a local segment. The write operations transfer data from a local segment to a remote segment.

The number of communication queues and their size can be configured at initialization time, otherwise default values will be used. The default values are implementation dependent. Maximum values are also defined.

For the write operation there are four different variants that allow different communication patterns:

- `gaspi_write`
- `gaspi_write_notify`
- `gaspi_write_list`
- `gaspi_write_list_notify`

The read operations have two different variants that allow different communication patterns:

- `gaspi_read`
- `gaspi_read_list`

The read operations do not support notification calls. This is due to the fact that a notification can only be transferred after ensuring that the communication request has been processed. This would imply that a subsequent wait call has to be invoked directly after invoking read. However, this can be managed more effectively by the application.

A valid one-sided communication request requires that the local and the remote segment are allocated, that there is a connection between the local and the remote GASPI process and that the remote segment has been registered on the local GASPI process.

8.2 Basic communication calls

8.2.1 `gaspi_write`

The simplest form of a write operation is `gaspi_write` which is a single communication call to write data to a remote location. It is an *asynchronous non-local time-based blocking* procedure.

```
GASPI_WRITE ( segment_id_local
            , offset_local
            , rank
            , segment_id_remote
            , offset_remote
            , size
            , queue
            , timeout )
```

Parameter:

(in) segment_id_local: the local segment ID to read from

(in) offset_local: the local offset in bytes to read from

(in) rank: the remote rank to write to

(in) segment_id_remote: the remote segment to write to

(in) offset_remote: the remote offset to write to

(in) size: the size of the data to write

(in) queue: the queue to use

(in) timeout: the timeout

```
gaspi_return_t
gaspi_write ( gaspi_segment_id_t segment_id_local
            , gaspi_offset_t offset_local
            , gaspi_rank_t rank
            , gaspi_segment_id_t segment_id_remote
            , gaspi_offset_t offset_remote
            , gaspi_size_t size
            , gaspi_queue_id_t queue
            , gaspi_timeout_t timeout )
```

```
function gaspi_write(segment_id_local,offset_local,&
&         rank, segment_id_remote,offset_remote,size,&
&         queue,timeout_ms) &
&         result( res ) bind(C, name="gaspi_write")
  integer(gaspi_segment_id_t), value :: segment_id_local
  integer(gaspi_offset_t), value :: offset_local
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_segment_id_t), value :: segment_id_remote
  integer(gaspi_offset_t), value :: offset_remote
  integer(gaspi_size_t), value :: size
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_write
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

gaspi_write posts a communication request which asynchronously transfers a contiguous block of *size* bytes from a source location of the local GASPI process to a target location of a remote GASPI process. This communication request is posted to the communication queue *queue*. The source location is specified by the pair *segment_id_local*, *offset_local*. The target location is specified by the triple *rank*, *segment_id_remote*, *offset_remote*.

A valid gaspi_write communication request requires that the local and the remote segment are allocated, that there is a connection between the local and the remote GASPI process and that the remote segment has been registered on the local GASPI process. Otherwise, the communication request is invalid and the procedure returns with GASPI_ERROR.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the communication request has been posted to the underlying network infrastructure. One new entry is inserted into the given queue.

Successive gaspi_write calls posted to the same queue and the same destination rank are not guaranteed to be non-overtaking. However, a subsequent gaspi_notify, which is posted to the same queue is guaranteed to be non-overtaking. In particular, one can hence assume, that if the corresponding notification has arrived on the remote process, the data from the earlier posted request to the same process has also arrived on the remote side.

gaspi_write calls may be posted from every thread of the GASPI process.

If the procedure returns with GASPI_TIMEOUT, the communication request could not be posted to the hardware during the given timeout. This can happen, if another thread is in a gaspi_wait for the same queue. A subsequent call of gaspi_write has to be invoked in order to complete the write call.

A communication request posted to a given queue can be considered as completed, if the correspondent gaspi_wait returns with GASPI_SUCCESS.

If the queue to which the communication request is posted is full, i. e. if the number of posted communication requests has already reached the queue size of a given queue, the communication request fails and the procedure returns with return value GASPI_ERROR. If a saturated queue is detected, there are the following two options: Either one invokes a gaspi_wait on the given queue in order to wait for all the posted requests to be finished. Alternatively it is possible to use another queue.

User advice: Return value GASPI_SUCCESS does not mean, that the data has been transferred or buffered or that the data has arrived at the remote side.
It is allowed to write data to the source location while the communication is ongoing. However, the result on the remote side would be some undefined interleaving of the data that was present when the call was issued and the data that was written later.
It is also allowed to read from the source location while the communcation is ongoing and such a read would retrieve the data written by the application.
Use gaspi_notify to synchronise the communication. ⌟

8.2.2 gaspi_read

The simplest form of a read operation is gaspi_read which is a single communication call to read data from a remote location. It is an *asynchronous non-local time-based blocking* procedure.

```
GASPI_READ ( segment_id_local
           , offset_local
           , rank
           , segment_id_remote
           , offset_remote
           , size
           , queue
           , timeout )
```

Parameter:

(in) segment_id_local: the local segment ID to write to

(in) offset_local: the local offset in bytes to write to

(in) rank: the remote rank to read from

(in) segment_id_remote: the remote segment to read from

(in) offset_remote: the remote offset to read from

(in) size: the size of the data to read

(in) queue: the queue to use

(in) timeout: the timeout

```
gaspi_return_t
gaspi_read ( gaspi_segment_id_t segment_id_local
           , gaspi_offset_t offset_local
           , gaspi_rank_t rank
           , gaspi_segment_id_t segment_id_remote
           , gaspi_offset_t offset_remote
           , gaspi_size_t size
           , gaspi_queue_id_t queue
           , gaspi_timeout_t timeout )
```

```
function gaspi_read(segment_id_local,offset_local,&
&         rank,segment_id_remote,offset_remote,size,&
&         queue,timeout_ms) &
&         result( res ) bind(C, name="gaspi_read")
  integer(gaspi_segment_id_t), value :: segment_id_local
  integer(gaspi_offset_t), value :: offset_local
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_segment_id_t), value :: segment_id_remote
  integer(gaspi_offset_t), value :: offset_remote
  integer(gaspi_size_t), value :: size
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_read
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_read` posts a communication request which asynchronously transfers a contiguous block of *size* bytes from a source location of a remote GASPI process to a target location of the local GASPI process. This communication request is posted to the communication queue *queue*. The target location is specified by the pair *segment_id_local*, *offset_local*. The source location is specified by the triple *rank*, *segment_id_remote*, *offset_remote*.

A valid `gaspi_read` communication request requires that the local and the remote segment are allocated, that there is a connection between the local and the remote GASPI process and that the remote segment has been registered on the local GASPI process. Otherwise, the communication request is invalid and the procedure returns with `GASPI_ERROR`.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the communication request has been posted to the underlying network infrastructure. One new entry is inserted into the given queue.

`gaspi_read` calls may be posted from every thread of the GASPI process.

If the procedure returns with `GASPI_TIMEOUT`, the communication request could not be posted to the hardware during the given timeout. This can happen, if another thread is in a `gaspi_wait` for the same queue. A subsequent call of `gaspi_read` has to be invoked in order to complete the read call.

A communication request posted to a given queue can be considered as completed, if the the correspondent `gaspi_wait` returns with `GASPI_SUCCESS`. For completed `gaspi_read` requests, the data is guaranteed to be locally available.

If the queue to which the communication request is posted is full, i. e. that the number of posted communication requests has already reached the queue size of a given queue, the communication request fails and the procedure returns with return value `GASPI_ERROR`. If a saturated queue is detected, there are the following two options: Either one invokes a `gaspi_wait` on the given queue in order to wait for all the posted requests to be finished. Or one tries to use another queue.

> *User advice:* Return value `GASPI_SUCCESS` does not mean, that the data transfer has started or that the data has been received at the local side. It is allowed to write data to the local target location while the communication is ongoing. However, the content of the memory would be some undefined interleaving of the data transferred from remote side and the data written locally.
> Also, it is allowed to read from the local target location while the communication is ongoing. Such a read would retrieve some undefined interleaving of the data that was present when the call was issued and the data that was transferred from the remote side.

8.2.3 gaspi_wait

The `gaspi_wait` procedure is a time-based blocking local procedure which waits until all one-sided communication requests posted to a given queue are processed by the network infrastructure. It is an *asynchronous non-local time-based blocking* procedure.

```
GASPI_WAIT ( queue
           , timeout )
```

Parameter:

(in) queue: the queue ID to wait for

(in) timeout: the timeout

```
gaspi_return_t
gaspi_wait ( gaspi_queue_id_t queue
           , gaspi_timeout_t timeout )
```

```
function gaspi_wait(queue,timeout_ms) &
&         result( res ) bind(C, name="gaspi_wait")
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_wait
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, the hitherto posted communication requests have been processed by the network infrastructure and the queue is cleaned up. After that, any communication request which has been posted to the given queue can be considered as completed on the local side.

gaspi_wait procedure calls may be posted from every thread of the local GASPI process. However, the wait operation is a thread exclusive operation and therefore needs privileged access to the queue which means that if a write/read is done while a wait is in operation, the write/read operation blocks to ensure correctness. Enforcing this provides correctness and safety to the user while being easier for the implementor and still allows for a high performance implementation. As a consequence, successive gaspi_wait calls invoked for the same queue by different threads are processed in some sequence one after another.

If the procedure returns with GASPI_TIMEOUT, the wait request could not be completed during the given timeout. This can happen, if there is another thread in a gaspi_wait for the same queue. A subsequent call of gaspi_wait has to be invoked in order to complete the call.

If the procedure returns with GASPI_ERROR, the wait request aborted abnormally.

In both cases, GASPI_TIMEOUT and GASPI_ERROR, the GASPI state vector should be checked in order to eliminate the possibility of a failure. If a failure is detected, all of the communication requests which have been posted to the given queue since the last gaspi_wait are in an undefined state. Here, undefined state means that the local GASPI process does not know which requests have been processed and which requests are still outstanding. A call to gaspi_queue_purge has to be invoked in order to reset the queue.

User advice: Return value GASPI_SUCCESS means, that the data of all posted write requests in this queue is in transfer to the remote side. It does not mean, that the data has arrived at the remote side. However, write accesses to the local source location will not affect the data that is placed in the remote target location. ⌟

User advice: Return value GASPI_SUCCESS means, that the data of all posted read requests have arrived at the local side. ⌟

8.2.4 Examples

Listing 11 shows a matrix transpose of a distributed square matrix implemented with the function gaspi_write.

Listing 11: GASPI all-to-all communication (matrix transpose) implemented with gaspi_write

```
#include <stdlib.h>
#include <GASPI.h>
#include <success_or_die.h>
#include <wait_if_queue_full.h>

extern void dump (int *arr, int nProc);

int
main (int argc, char *argv[])
{
  ASSERT (gaspi_proc_init (GASPI_BLOCK));

  gaspi_rank_t iProc;
  gaspi_rank_t nProc;

  ASSERT (gaspi_proc_rank (&iProc));
  ASSERT (gaspi_proc_num (&nProc));

  gaspi_notification_id_t notification_max;
  ASSERT (gaspi_notification_num(&notification_max));

  if (notification_max < (gaspi_notification_id_t)nProc)
    {
      exit (EXIT_FAILURE);
    }

  ASSERT (gaspi_group_commit (GASPI_GROUP_ALL, GASPI_BLOCK));

  const gaspi_segment_id_t segment_id_src = 0;
  const gaspi_segment_id_t segment_id_dst = 1;

  const gaspi_size_t segment_size = nProc * sizeof(int);

```

```
ASSERT (gaspi_segment_create ( segment_id_src, segment_size
                             , GASPI_GROUP_ALL, GASPI_BLOCK
                             , GASPI_ALLOC_DEFAULT
                             )
       );
ASSERT (gaspi_segment_create ( segment_id_dst, segment_size
                             , GASPI_GROUP_ALL, GASPI_BLOCK
                             , GASPI_ALLOC_DEFAULT
                             )
       );

int *src = NULL;
int *dst = NULL;

ASSERT (gaspi_segment_ptr (segment_id_src, &src));
ASSERT (gaspi_segment_ptr (segment_id_dst, &dst));

const gaspi_queue_id_t queue_id = 0;

for (gaspi_rank_t rank = 0; rank < nProc; ++rank)
  {
    src[rank] = iProc * nProc + rank;

    const gaspi_offset_t offset_src = rank * sizeof (int);
    const gaspi_offset_t offset_dst = iProc * sizeof (int);
    const gaspi_notification_id_t notify_ID = rank;

    wait_if_queue_full (queue_id, 2);

    const gaspi_notification_t notify_val = 1;

    ASSERT
      (gaspi_write_notify ( segment_id_src, offset_src
                          , rank, segment_id_dst, offset_dst
                          , sizeof (int), notify_ID, notify_val
                          , queue_id, GASPI_BLOCK
                          )
      );
  }

gaspi_notification_id_t notify_cnt = nProc;
gaspi_notification_id_t first_notify_id;

while (notify_cnt > 0)
{
  ASSERT (gaspi_notify_waitsome ( segment_id_dst, 0, nProc,
                                , &first_notify_id, GASPI_BLOCK));

  gaspi_notification_id_t notify_val = 0;

```

```
    ASSERT (gaspi_notify_reset (segment_id_dst, first_notify_id
                              , &notify_val));

    if (notify_val != 0)
      {
        --notify_cnt;
      }
  }

  dump (dst, nProc);

  ASSERT (gaspi_wait (queue_id, GASPI_BLOCK));

  ASSERT (gaspi_barrier (GASPI_GROUP_ALL, GASPI_BLOCK));

  ASSERT (gaspi_proc_term (GASPI_BLOCK));

  return EXIT_SUCCESS;
}
```

Listing 12 shows a matrix transpose of a distributed square matrix implemented with the function `gaspi_read`. Please note the differences between the transpose implemented with write and the transpose implemented with read: The implementation using write can initialize the matrix on-the-fly, right before the data is transferred, while the implementation using read has to synchronise all processes after the local initialization in order to be sure to read valid data. On the other hand, in the implementation using write one has to synchronise after the local wait whereas in the implementation using read one can directly use the data after the local wait returns.

Listing 12: GASPI all-to-all communication (matrix transpose) implemented with `gaspi_read`

```
#include <stdlib.h>
#include <GASPI.h>
#include <success_or_die.h>
#include <wait_if_queue_full.h>

extern void dump (int *arr, int nProc);

int
main (int argc, char *argv[])
{
  ASSERT (gaspi_proc_init (GASPI_BLOCK));

  gaspi_rank_t iProc;
  gaspi_rank_t nProc;

  ASSERT (gaspi_proc_rank (&iProc));
  ASSERT (gaspi_proc_num (&nProc));

```

```
ASSERT (gaspi_group_commit (GASPI_GROUP_ALL, GASPI_BLOCK));

const gaspi_segment_id_t segment_id_src = 0;
const gaspi_segment_id_t segment_id_dst = 1;

const gaspi_size_t segment_size = nProc * sizeof(int);

ASSERT (gaspi_segment_create ( segment_id_src, segment_size
                             , GASPI_GROUP_ALL, GASPI_BLOCK
                             , GASPI_ALLOC_DEFAULT
                             )
       );
ASSERT (gaspi_segment_create ( segment_id_dst, segment_size
                             , GASPI_GROUP_ALL, GASPI_BLOCK
                             , GASPI_ALLOC_DEFAULT
                             )
       );

int *src = NULL;
int *dst = NULL;

ASSERT (gaspi_segment_ptr (segment_id_src, &src));
ASSERT (gaspi_segment_ptr (segment_id_dst, &dst));

const gaspi_queue_id_t queue_id = 0;

for (gaspi_rank_t rank = 0; rank < nProc; ++rank)
  {
    src[rank] = iProc * nProc + rank;
  }

ASSERT (gaspi_barrier (GASPI_GROUP_ALL, GASPI_BLOCK));

for (gaspi_rank_t rank = 0; rank < nProc; ++rank)
  {
    const gaspi_offset_t offset_src = iProc * sizeof (int);
    const gaspi_offset_t offset_dst = rank * sizeof (int);

    wait_if_queue_full (queue_id, 1);

    ASSERT (gaspi_read ( segment_id_dst, offset_dst
                       , rank, segment_id_src, offset_src
                       , sizeof (int), queue_id, GASPI_BLOCK
                       )
           );
  }

ASSERT (gaspi_wait (queue_id, GASPI_BLOCK));

dump (dst, nProc);
```

```
  ASSERT (gaspi_barrier (GASPI_GROUP_ALL, GASPI_BLOCK));

  ASSERT (gaspi_proc_term (GASPI_BLOCK));

  return EXIT_SUCCESS;
}
```

The definition of the macro `ASSERT` is given in the listings 17 and 18. The definition of the function `wait_if_queue_full` is given in the listings 19 and 20 starting on page 131.

8.3 Weak synchronisation primitives

8.3.1 Introduction

The one-sided communication procedures have the characteristics that the entire communication is managed by the local process only. The remote process is not involved. This has the advantage that there is no inherent synchronisation between the local and the remote process in every communication request. However, at some point, the remote process needs the information as to whether the data which has been sent to that process has arrived and is valid.

To this end GASPI provides so-called weak synchronisation primitives which allows the application to inform the remote side that the data has been transferred by updating a notification on the remote side. These notifications must be submitted to the same queue to which the data payload has been attached. Otherwise, causality is not guaranteed.

As counterpart, there are routines which wait for an update of a single or even an entire set of notifications. There is a thread safe atomic function to reset the local notification with a given ID which returns the value of the notification before it is reset.

These notification procedures are also one-sided and involve only the local process.

8.3.2 `gaspi_notify`

`gaspi_notify` is an *asynchronous non-local time-based blocking* procedure.

```
GASPI_NOTIFY ( segment_id
             , rank
             , notification_id
             , notification_value
             , queue
             , timeout )
```

Parameter:

(in) segment_id: the remote segment bound to the notification

(in) rank: the remote rank to notify

(in) notification_id: the remote notification ID

(in) notification_value: the notification value (> 0) to write

(in) queue: the queue to use

(in) timeout: the timeout

```
gaspi_return_t
gaspi_notify ( gaspi_segment_id_t segment_id
             , gaspi_rank_t rank
             , gaspi_notification_id_t notification_id
             , gaspi_notification_t notification_value
             , gaspi_queue_id_t queue
             , gaspi_timeout_t timeout )
```

```
function gaspi_notify(segment_id_remote,rank,notification_id, &
&         notification_value,queue,timeout_ms) &
&         result( res ) bind(C, name="gaspi_notify")
  integer(gaspi_segment_id_t), value :: segment_id_remote
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_notification_id_t), value :: notification_id
  integer(gaspi_notification_t), value :: notification_value
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_notify
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_notify` posts a notification request which asynchronously transfers the notification *notification_value* of the local GASPI process to an internal notification buffer of a remote GASPI process. This notification request is posted to the communication queue *queue*. The remote notification buffer is specified by the pair *rank*, *notification_id*.

A valid `gaspi_notify` communication request requires that there is a connection between the local and the remote GASPI process. Otherwise, the communication request is invalid and the procedure returns with `GASPI_ERROR`.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the notification request has been posted to the underlying network infrastructure.

A gaspi_notify call which is posted subsequent to an arbitrary number of gaspi_write requests and which is posted to the same queue and the same destination rank is guaranteed to be non-overtaking. Non-overtaking means that the order of communication requests is preserved on the remote side. In particular, one can assume, that if the data from the gaspi_notify request has arrived on the remote process, also the data from the earlier posted write request(s) to the same process have arrived on the remote side.

gaspi_notify calls may be posted from every thread of the GASPI process.

If the procedure returns with GASPI_TIMEOUT, the notification request could not be posted to the hardware during the given timeout. This can happen if another thread is in a gaspi_wait for the same queue. A subsequent call of gaspi_notify has to be invoked in order to complete the call.

A notification request posted to a given queue can be considered as completed, if the the correspondent gaspi_wait returns with GASPI_SUCCESS.

If the queue to which the communication request is posted is full, i. e. that the number of posted communication requests has already reached the queue size of a given queue, the communication request fails.

> *User advice:* Return value GASPI_SUCCESS does not mean, that the notification has been transferred or that the notification has arrived at the remote side. ⌟

8.3.3 gaspi_notify_waitsome

For the procedures with notification, gaspi_notify and the extendend functions gaspi_write_notify and gaspi_read_notify, gaspi_notify_waitsome is the correspondent wait procedure for the notified receiver side (which is remote for the functions gaspi_notify and gaspi_write_notify and local for the function gaspi_read_notify). gaspi_notify_waitsome is a *synchronous, non-local time-based blocking* procedure.

```
GASPI_NOTIFY_WAITSOME ( segment_id
                      , notification_begin
                      , notification_num
                      , first_id
                      , timeout )
```

Parameter:

(in) segment_id: the segment bound to the notification

(in) notification_begin: the local notification ID for the first notification to wait for

(in) notification_num: the number of notification ID's to wait for

(out) first_id: the id of the first notification that arrived

(in) timeout: the timeout

```
gaspi_return_t
gaspi_notify_waitsome ( gaspi_segment_id_t segment_id
                      , gaspi_notification_id_t notific_begin
                      , gaspi_number_t notification_num
                      , gaspi_notification_id_t *first_id
                      , gaspi_timeout_t timeout )
```

```
function gaspi_notify_waitsome(segment_id_local,&
&         notification_begin,num,first_id,timeout_ms) &
&         result( res ) bind(C, name="gaspi_notify_waitsome")
  integer(gaspi_segment_id_t), value :: segment_id_local
  integer(gaspi_notification_id_t), value :: notification_begin
  integer(gaspi_number_t), value :: num
  integer(gaspi_notification_id_t) :: first_id
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_notify_waitsome
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

gaspi_notify_waitsome waits that at least one of a number of consecutive notifications residing in the local internal buffer has a value that is not zero.

The notification buffer is specified by the pair *notification_begin*, *notification_num*. It contains *notification_num* many consecutive notifications beginning at the notification with ID *notification_begin*.

If *notification_num* == 0 then gaspi_notify_waitsome returns immediately with GASPI_SUCCESS.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the value of at least one of the notifications in the notification buffer has changed to a value that is not zero. All threads that are waiting for the notifications are notified.

If the procedure returns with GASPI_TIMEOUT, no notification has changed during the given period of time.

In case of an error, i. e. GASPI_ERROR, the values of the notifications are undefined.

User advice: One scenario for the usage of `gaspi_notify_waitsome` inspecting only one notification is the following: The remote side uses a `gaspi_write` call followed by a subsequent call of `gaspi_notify` posted to the same queue and the same destination rank. GASPI guarantees, that if the notification has arrived on the remote process, the previously posted request carrying the work load has arrived as well. ⌟

User advice: One scenario for the usage of `gaspi_notify_waitsome` inspecting only one notification is the following: The local side posts a `gaspi_read_notify` call. GASPI guarantees, that if the notification has arrived on the local process, the posted read request carrying the work load of the function `gaspi_gaspi_read_notify` has arrived as well. ⌟

User advice: If in a multi-threaded application more than one thread calls `gaspi_notify_waitsome` for the range of notifications, then all waiting threads are notified about the change of at least one of the notifications. By inspecting the actual values of each of the notifications with `gaspi_notify_reset`, only one thread per changed notification receives a value different from zero. ⌟

User advice: In a multi-threaded application the code in listing 13 selects one thread to act on the change of a single notification. The code waits in a blocking manner and thus cannot be used in failure tolerant applications. ⌟

Listing 13: Blocking waitsome in a multi-threaded application

```
#include <GASPI.h>
#include <success_or_die.h>

extern void process ( const gaspi_notification_id_t id
                    , const gaspi_notification_t val
                    );

void blocking_waitsome ( const gaspi_notification_id_t id_begin
                       , const gaspi_notification_id_t id_end
                       , const gaspi_segment_id_t seg_id
                       )
{
  gaspi_notification_id_t first_id;

  ASSERT ( gaspi_notify_waitsome ( seg_id
                                 , id_begin
                                 , id_end - id_begin
                                 , &first_id
                                 , GASPI_BLOCK
                                 )
         );

  gaspi_notification_t val = 0;
```

```

  // atomic reset
  ASSERT (gaspi_notify_reset (seg_id, first_id, &val));

  // other threads are notified too!
  process (first_id, val);
}
```

8.3.4 gaspi_notify_reset

For the `gaspi_notify_waitsome` procedure, there is a notification initialization procedure which resets the given notification to zero. It is a *synchronous local blocking* procedure.

```
GASPI_NOTIFY_RESET ( segment_id
                   , notification_id
                   , old_notification_val )
```

Parameter:

(in) segment_id: the segment bound to the notification

(in) notification_id: the local notification ID to reset

(out) old_notification_val: notification value before reset

```
gaspi_return_t
gaspi_notify_reset ( gaspi_segment_id_t segment_id
                   , gaspi_notification_id_t notification_id
                   , gaspi_notification_t *old_notification_val)
```

```
function gaspi_notify_reset(segment_id_local, &
&         notification_id,old_notification_val) &
&         result( res ) bind(C, name="gaspi_notify_reset")
  integer(gaspi_segment_id_t), value :: segment_id_local
  integer(gaspi_notification_id_t), value :: notification_id
  integer(gaspi_notification_t) :: old_notification_val
  integer(gaspi_return_t) :: res
end function gaspi_notify_reset
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error `gaspi_notify_reset` resets the notification with ID *notification_id* to zero. The function `gaspi_notify_reset` is an atomic operation: Threads can use `gaspi_notify_reset` to safely extract the value of a specific notification.

The notification buffer on the local side is specified by the notification ID *notification_id*.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the value of the notification buffer was set to zero and *old_notification_val* contains the content of the notification buffer before it was set to zero. To read the old value and to set the value to zero is a single atomic operation.

`gaspi_notify_reset` calls may be posted from every thread of the GASPI process.

In case of error, i. e. return value `GASPI_ERROR`, the value of *old_notification_val* is undefined.

8.4 Extended communication calls

All restrictions applying to `gaspi_write` and `gaspi_notify` also apply here. In case of timeout or error, no assumptions may be made regarding either the written data or the notification.

8.4.1 `gaspi_write_notify`

The `gaspi_write_notify` variant extends the simple `gaspi_write` with a notification on the remote side. This applies to communication patterns that require tighter synchronisation on data movement. The remote receiver of the data is notified when the write is finished and can verify this through the respective wait procedure. It is an *asynchronous non-local time-based blocking* procedure.

```
GASPI_WRITE_NOTIFY ( segment_id_local
                   , offset_local
                   , rank
                   , segment_id_remote
                   , offset_remote
                   , size
                   , notification_id
                   , notification_value
                   , queue
                   , timeout )
```

Parameter:

(in) segment_id_local: the local segment ID to read from

(in) offset_local: the local offset in bytes to read from

(in) rank: the remote rank to write to

(in) segment_id_remote: the remote segment to write to

(in) offset_remote: the remote offset to write to

(in) size: the size of the data to write

(in) notification_id: the remote notification ID

(in) notification_value: the value of the notification to write

(in) queue: the queue to use

(in) timeout: the timeout

```
gaspi_return_t
gaspi_write_notify ( gaspi_segment_id_t segment_id_local
                   , gaspi_offset_t offset_local
                   , gaspi_rank_t rank
                   , gaspi_segment_id_t segment_id_remote
                   , gaspi_offset_t offset_remote
                   , gaspi_size_t size
                   , gaspi_notification_id_t notification_id
                   , gaspi_notification_t notification_value
                   , gaspi_queue_id_t queue
                   , gaspi_timeout_t timeout )
```

```
function gaspi_write_notify(segment_id_local,offset_local,&
&         rank,segment_id_remote,offset_remote,size,&
&         notification_id,notification_value,queue,&
&         timeout_ms) &
&         result( res ) bind(C, name="gaspi_write_notify")
  integer(gaspi_segment_id_t), value :: segment_id_local
  integer(gaspi_offset_t), value :: offset_local
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_segment_id_t), value :: segment_id_remote
  integer(gaspi_offset_t), value :: offset_remote
  integer(gaspi_size_t), value :: size
  integer(gaspi_notification_id_t), value :: notification_id
  integer(gaspi_notification_t), value :: notification_value
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_write_notify
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

Implementor advice: The procedure is not semantically equivalent to a call to `gaspi_write` and a subsequent call of `gaspi_notify`. This call does not enforce an ordering relative to other write operations. ⌟

8.4.2 gaspi_write_list

The `gaspi_write_list` variant allows strided communication where a list of different data locations are processed at once. Semantically, it is equivalent to a sequence of calls to `gaspi_write` but it should (if possible) be more efficient. It is an *asynchronous non-local time-based blocking* procedure.

```
GASPI_WRITE_LIST ( num
                 , segment_id_local[num]
                 , offset_local[num]
                 , rank
                 , segment_id_remote[num]
                 , offset_remote[num]
                 , size[num]
                 , queue
                 , timeout )
```

Parameter:

(in) num: the number of elements to write

(in) segment_id_local[num]: list of local segment ID's to read from

(in) offset_local[num]: list of local offsets in bytes to read from

(in) rank: the remote rank to write to

(in) segment_id_remote[num]: list of remote segments to write to

(in) offset_remote[num]: list of remote offsets to write to

(in) size[num]: list of sizes of the data to write

(in) queue: the queue to use

(in) timeout: the timeout

```
gaspi_return_t
gaspi_write_list ( gaspi_number_t num
                 , gaspi_segment_id_t const *segment_id_local
                 , gaspi_offset_t const *offset_local
                 , gaspi_rank_t rank
                 , gaspi_segment_id_t const *segment_id_remote
                 , gaspi_offset_t const *offset_remote
                 , gaspi_size_t const *size
                 , gaspi_queue_id_t queue
                 , gaspi_timeout_t timeout )
```

```
function gaspi_write_list(num,segment_id_local,offset_local,&
&         rank,segment_id_remote,offset_remote,size,queue,&
&         timeout_ms) &
&         result( res ) bind(C, name="gaspi_write_list")
  integer(gaspi_number_t), value :: num
  type(c_ptr), value :: segment_id_local
  type(c_ptr), value  :: offset_local
  integer(gaspi_rank_t), value :: rank
  type(c_ptr), value :: segment_id_remote
  type(c_ptr), value :: offset_remote
  type(c_ptr), value :: size
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_write_list
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

> *Implementor advice:* The procedure is semantically equivalent to *num* subsequent calls of `gaspi_write` with the given local and remote location specification, provided that the destination rank and the used queue are invariant. However, it should be implemented more efficiently, if supported by the network infrastructure. ⌟

8.4.3 `gaspi_write_list_notify`

The `gaspi_write_list_notify` operation performs strided communication as `gaspi_write_list` but also includes a notification that the remote receiver can use to ensure that the communication step is completed. It is an *asynchronous non-local time-based blocking* procedure.

```
GASPI_WRITE_LIST_NOTIFY
                ( num
                , segment_id_local[num]
                , offset_local[num]
                , rank
                , segment_id_remote[num]
                , offset_remote[num]
                , size[num]
                , notification_id
                , notification_value
                , queue
                , timeout )
```

Parameter:

(in) num: the number of elements to write

(in) segment_id_local[num]: list of local segment ID's to read from

(in) offset_local[num]: list of local offsets in bytes to read from

(in) rank: the remote rank to be write to

(in) segment_id_remote[num]: list of remote segments to write to

(in) offset_remote[num]: list of remote offsets to write to

(in) size[num]: list of sizes of the data to write

(in) notification_id: the remote notification ID

(in) notification_value: the value of the notification to write

(in) queue: the queue to use

(in) timeout: the timeout

```
gaspi_return_t
gaspi_write_list_notify
                ( gaspi_number_t num
                , gaspi_segment_id_t const *segment_id_local
                , gaspi_offset_t const *offset_local
                , gaspi_rank_t rank
                , gaspi_segment_id_t const *segment_id_remote
                , gaspi_offset_t const *offset_remote
                , gaspi_size_t const *size
                , gaspi_notification_id_t notification_id
                , gaspi_notification_t notification_value
                , gaspi_queue_id_t queue
                , gaspi_timeout_t timeout )
```

```
function gaspi_write_list_notify(num,segment_id_local,&
&         offset_local,rank,segment_id_remote,&
&         offset_remote,size,segment_id_notification, &
&         notification_id,notification_value,queue,timeout_ms) &
&         result( res ) bind(C, name="gaspi_write_list_notify")
  integer(gaspi_number_t), value :: num
  type(c_ptr), value :: segment_id_local
  type(c_ptr), value :: offset_local
  integer(gaspi_rank_t), value :: rank
  type(c_ptr), value :: segment_id_remote
  type(c_ptr), value :: offset_remote
  type(c_ptr), value :: size
  integer(gaspi_segment_id_t), value :: segment_id_notification
  integer(gaspi_notification_id_t), value :: notification_id
  integer(gaspi_notification_t), value :: notification_value
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_write_list_notify
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

> *Implementor advice:* The procedure is not semantically equivalent to a call to `gaspi_write_list` and a subsequent call of `gaspi_notify`. This call does not enforce an ordering relative to other write operations. ⌟

8.4.4 `gaspi_read_notify`

The `gaspi_read_notify` variant extends the simple `gaspi_read` with a notification on the local side. This applies to communication patterns that require tighter synchronisation on data movement. The local receiver of the data is notified when the read is finished and can verify this through the procedure `gaspi_waitsome`. It is an *asynchronous non-local time-based blocking* procedure.

```
GASPI_READ_NOTIFY (  segment_id_local
                   , offset_local
                   , rank
                   , segment_id_remote
                   , offset_remote
                   , size
                   , notification_id_local
                   , queue
                   , timeout )
```

Parameter:

(in) segment_id_local: the local segment to write to

(in) offset_local: the local offset to write to

(in) rank: the remote rank to read from

(in) segment_id_remote: the remote segment ID to read from

(in) offset_remote: the remote offset in bytes to read from

(in) size: the size of the data to read

(in) notification_id: the local notification ID

(in) queue: the queue to use

(in) timeout: the timeout

```
gaspi_return_t
gaspi_read_notify (  gaspi_segment_id_t segment_id_local
                   , gaspi_offset_t offset_local
                   , gaspi_rank_t rank
                   , gaspi_segment_id_t segment_id_remote
                   , gaspi_offset_t offset_remote
                   , gaspi_size_t size
                   , gaspi_notification_id_t notification_id
                   , gaspi_queue_id_t queue
                   , gaspi_timeout_t timeout )
```

```
function gaspi_read_notify(segment_id_local,offset_local,rank,&
&         segment_id_remote, offset_remote,&
&         size,notification_id,queue,&
&         timeout_ms) &
&         result( res ) bind(C, name="gaspi_read_notify")
  integer(gaspi_segment_id_t), value :: segment_id_local
  integer(gaspi_offset_t), value :: offset_local
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_segment_id_t), value :: segment_id_remote
  integer(gaspi_offset_t), value :: offset_remote
  integer(gaspi_size_t), value :: size
  integer(gaspi_notification_id_t), value :: notification_id
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_read_notify
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

> *User advice:* In contrast to the procedure `gaspi_write_notify`, the notification in the procedure `gaspi_read_notify` carries the (fixed) notification value of 1. Similar to the procedure `gaspi_write_notify` a call to `gaspi_read_notify` only guarantees ordering with respect to the data bundled in this communication and the given notification. Specifically there are no ordering guarantees to other read operations. For this latter functionality a call to the `gaspi_wait` procedure is required. ⌟

> *User advice:* The two GASPI functions `gaspi_read_notify` and `gaspi_waitsome` establish a logical and thread safe happens-before relation between them. ⌟

User advice: The notifcation driven `gaspi_read_notify` complements the `gaspi_write_notify` functionality.
While a `gaspi_read_notify` features a variety of use cases (e.g. in distributed memory management) one of the more remarkable goals of the function `gaspi_read_notify` is to establish latency-tolerant multithreading in distributed memory systems. To that end we first note that GASPI is able to sustain an extremely high concurrency: the number of messages GASPI can keep in flight at any point in time is (in first order) given by the product of the number of available queues and the queue depth ($queue\_num * queue\_size\_max$).
Following ideas which go back to the first of Cray's MTA machines, we hence can leverage Little's law ($bandwidth = concurrency/latency$) and use the high concurrency available in GASPI to effectively hide away latency for remote read access in distributed memory systems. In doing so we gain e.g. the ability to perform overhead-free graph traversal for non-partitionable (but distributed) large-scale graphs. We note that the same general principle holds true for all applications, which allow for a high concurrency: whenever we can sustain high concurrency in fetching and evaluating remote data, Little's law will allow us to tolerate the corresponding read latency. This applies to all forms of parallel graph-problems, parallel table lookups, parallel searches in a data-base and many other use cases.

⌟

Listing 14: `gaspi_read_notify` Example usage

```
// Pipelined read and processing of data
// The pipeline consists of the following two stages
// 1. Read remote data with a predefined number of chunks
// 2. Perform multithreaded waitsome, subsequent processing of
//    the data chunks, and a consecutive read_notify in order to
//    sustain the pipeline.

#include <GASPI.h>
#include <success_or_die.h>

extern void process( gaspi_segment_id_t segment_id_local
                   , gaspi_offset_t offset_local
                   , gaspi_size_t size
                   , gaspi_notification_id_t id
                   );

// Note: For sake of simplicity we have omitted checking
//       the number of used chunks vs. the actually available
//       notification ressources as well as properly checking the
//       queue status. (see e.g. example for gaspi_wait,
//       wait_if_queue_full())

void pipelined_read_and_process( int num_chunks
                               , gaspi_segment_id_t segment_id_local
                               , gaspi_offset_t offset_local
```

```
                      , gaspi_rank_t rank
                      , gaspi_segment_id_t segment_id_remote
                      , gaspi_offset_t offset_remote
                      , gaspi_size_t chunk_size
                      , gaspi_queue_id_t queue_id
                      )
{
  const int nthreads = omp_get_max_threads();
  const int num_initial_chunks = nthreads * 4;
  int i;

  // Start GASPI accumulate pipeline
  for (i = 0; i < num_initial_chunks; ++i)
  {
    ASSERT (gaspi_read_notify (segment_id_local
                              , (offset_local+i*chunk_size)
                              , rank
                              , segment_id_remote
                              , (offset_remote+i*chunk_size)
                              , chunk_size
                              , i
                              , queue_id
                              , GASPI_BLOCK ));
  }

#pragma omp parallel
  {
    int const tid = omp_get_thread_num();

    // For sake of simplicity we use notifications
    // which are exclusive per thread.

    gaspi_notification_id_t id, first = tid;
    gaspi_notification_id_t next  = first + num_initial_chunks;

    while(first < num_chunks)
    {
      ASSERT (gaspi_notify_waitsome ( segment_id_local,
                                    , first
                                    , 1
                                    , &id
                                    , GASPI_BLOCK));

      gaspi_notification_t val = 0;
      ASSERT (gaspi_notify_reset (segment_id_local
                                    , id
                                    , &val));

      // process received data chunk
      process( segment_id_local
```

```
              , (offset_local+id*chunk_size)
              , chunk_size
              , id
              );

      first += nthreads;
      next  += nthreads;

      if (next < num_chunks)
      {
        // start next read, sustain pipeline.
        ASSERT (gaspi_read_notify (segment_id_local
                                 , (offset_local+next*chunk_size)
                                 , rank
                                 , segment_id_remote
                                 , (offset_remote+next*chunk_size)
                                 , chunk_size
                                 , next
                                 , queue_id
                                 , GASPI_BLOCK ));
      }
    }

}
```

Implementor advice: The procedure is not semantically equivalent to a call to `gaspi_read` and a subsequent call of `gaspi_notify`, since the latter aims at remote completion rather than local completion. Also this call does not enforce an ordering relative to other read operations. We note that the procedure `gaspi_read_notify` aims at massive concurrency rather than minimal read latency, hence it should be implemented accordingly. ⌟

8.4.5 gaspi_read_list

The `gaspi_read_list` variant allows strided communication where a list of different data locations are processed at once. Semantically, it is equivalent to a sequence of calls to `gaspi_read` but it should (if possible) be more efficient. It is an *asynchronous non-local time-based blocking* procedure.

```
GASPI_READ_LIST ( num
                , segment_id_local[num]
                , offset_local[num]
                , rank
                , segment_id_remote[num]
                , offset_remote[num]
                , size[num]
                , queue
                , timeout )
```

Parameter:

(in) num: the number of elements to read

(in) segment_id_local[num]: list of local segment ID's to write to

(in) offset_local[num]: list of local offsets in bytes to write to

(in) rank: the remote rank to read from

(in) segment_id_remote[num]: list of remote segments to read from

(in) offset_remote[num]: list of remote offsets to read from

(in) size[num]: list of sizes of the data to read

(in) queue: the queue to use

(in) timeout: the timeout

```
gaspi_return_t
gaspi_read_list ( gaspi_number_t num
                , gaspi_segment_id_t const *segment_id_local
                , gaspi_offset_t const *offset_local
                , gaspi_rank_t rank
                , gaspi_segment_id_t const *segment_id_remote
                , gaspi_offset_t const *offset_remote
                , gaspi_size_t const *size
                , gaspi_queue_id_t queue
                , gaspi_timeout_t timeout )
```

```
function gaspi_read_list(num,segment_id_local,offset_local,&
&         rank,segment_id_remote,offset_remote,size,queue,&
&         timeout_ms) &
&         result( res ) bind(C, name="gaspi_read_list")
  integer(gaspi_number_t), value :: num
  type(c_ptr), value :: segment_id_local
  type(c_ptr), value :: offset_local
  integer(gaspi_rank_t), value :: rank
  type(c_ptr), value :: segment_id_remote
  type(c_ptr), value :: offset_remote
  type(c_ptr), value :: size
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_read_list
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error

8.5 Communication utilities

8.5.1 gaspi_queue_create

The gaspi_queue_create procedure is a *synchronous non-local time-based blocking* procedure which creates a new queue for communication.

```
GASPI_QUEUE_CREATE ( queue
                   , timeout
                   )
```

Parameter:

(out) queue: the created queue

(in) timeout: the timeout

```
gaspi_return_t
gaspi_queue_create ( gaspi_queue_id_t queue
                   , gaspi_timeout_t timeout
                   )
```

```
function gaspi_queue_create (queue, timeout) &
&        result(res) bind (C, name="gaspi_queue_create" )
  integer(gaspi_queue_id_t) :: queue
  integer(gaspi_timeout_t), value :: timeout
  integer(gaspi_return_t) :: res
end function gaspi_queue_create
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error

After successful procedure completion, i. e. return value GASPI_SUCCESS, the communication *queue* is created and available for communication requests on it.

If the procedure returns with GASPI_TIMEOUT, the creation request could not be completed during the given timeout. A subsequent call to gaspi_queue_create has to be performed in order to complete the queue creation request.

If the procedure returns with GASPI_ERROR, the queue creation failed. Attempts to post requests in the queue result in undefined behaviour.

> *User advice:* The lifetime of a created queue should be kept as long as possible, avoiding repeated cycles of creation/deletion of a queue. ⌟

> *Implementor advice:* The maximum number of allowed queues may be limited in order to keep resources requirements low. ⌟

> *Implementor advice:* The communication infrastructure must be respected i. e. previously established connections (e. g. invoking gaspi_connect) must be able to use the newly created queue. ⌟

8.5.2 gaspi_queue_delete

The gaspi_queue_delete procedure is a *synchronous non-local time-based blocking* procedure which deletes a given queue.

```
GASPI_QUEUE_DELETE ( queue )
```

Parameter:

(in) queue: the queue to delete

```
gaspi_return_t
gaspi_queue_delete ( gaspi_queue_id_t queue )
```

```
function gaspi_queue_delete ( queue ) &
&        result(res) bind (C, name="gaspi_queue_delete" )
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_return_t) :: res
end function gaspi_queue_delete
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error ⌟

After successful procedure completion, i. e. return value GASPI_SUCCESS, the communication *queue* is deleted and no longer available for communication. It is an application error to use the queue after gaspi_queue_delete has been invoked.

If the procedure returns with GASPI_ERROR, the delete request failed.

> *User advice:* The procedure gaspi_wait should be invoked before deleting a queue in order to ensure that all posted requests (if any) are completed. ⌟

8.5.3 gaspi_queue_size

The gaspi_queue_size procedure is a *synchronous local blocking* procedure which determines the number of open communication requests posted to a given queue.

```
GASPI_QUEUE_SIZE ( queue
                 , queue_size )
```

Parameter:

(in) queue: the queue to probe

(out) queue_size: the number of open requests posted to the queue

```
gaspi_return_t
gaspi_queue_size ( gaspi_queue_id_t queue
                 , gaspi_number_t const *queue_size )
```

```
function gaspi_queue_size(queue,queue_size) &
&         result( res ) bind(C, name="gaspi_queue_size")
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_number_t) :: queue_size
  integer(gaspi_return_t) :: res
end function gaspi_queue_size
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

After successful procedure completion, i. e. return value GASPI_SUCCESS, the parameter *queue_size* contains the number of open requests posted to the queue *queue*. In a threaded program this result is uncertain, since another thread may have posted an additional request in the meantime or issued a wait call.

The queue size is set to zero by a successful call to gaspi_wait.

In case of error, the return value is GASPI_ERROR. The parameter *queue_size* has an undefined value.

8.5.4 gaspi_queue_purge

The gaspi_queue_purge procedure is a *synchronous local time-based blocking* procedure which purges a given queue.

```
GASPI_QUEUE_PURGE ( queue
                  , timeout )
```

Parameter:

(in) queue: the queue to purge

(in) timeout: the timeout

```
gaspi_return_t
gaspi_queue_purge ( gaspi_queue_id_t queue
                  , gaspi_timeout_t timeout )
```

```
function gaspi_queue_purge(queue,timeout) &
&         result( res ) bind(C, name="gaspi_queue_purge")
  integer(gaspi_queue_id_t), value :: queue
  integer(gaspi_timeout_t), value :: timeout
  integer(gaspi_return_t) :: res
end function gaspi_queue_purge
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

This procedure should only be invoked in the situation in which a node failure is detected by inspecting the global health state with gaspi_state_vec_get.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the communication *queue* is purged. All communication requests posted to the queue *queue* are eliminated from the queue. The local GASPI process has no information about the completion of communication requests posted to the given queue since the last invocation of gaspi_wait.

If the procedure returns with GASPI_TIMEOUT, the purge request could not be completed during the given timeout. This might happen if there is another thread in a gaspi_wait for the same queue. A subsequent call of gaspi_queue_purge has to be invoked in order to complete the call.

If the procedure returns with GASPI_ERROR, the purge request aborted abnormally.

9 Passive communication

9.1 Introduction and overview

Passive communication has a two-sided semantic, where there is a matching receiver to a send request. Passive communication aims at communication patterns where the sender is unknown (i. e. it can be any process from the receiver perspective) but there is potentially the need for synchronisation between processes. Typical example uses cases are:

- Distributed update where many processes contribute to the data of one process.
- Pass arguments and results.
- Global error handling.

The implementation should try to enforce fairness in communication that is, no sender should see its communication request delayed indefinitely.

The passive keyword means that the communication calls should avoid busy-waiting and consume no CPU cycles, freeing the system for computation.

Both the send and the matching receive are *time-based blocking*. A valid passive communication request requires that the local and the remote segment are allocated and that there is a connection between the local and the remote GASPI process. Otherwise, the communication request is invalid and the procedure returns with `GASPI_ERROR`.

9.2 Passive communication calls

9.2.1 gaspi_passive_send

`gaspi_passive_send` is the routine called by the sender side to engage in passive communication. It is an *synchronous non-local time-based blocking* procedure.

```
GASPI_PASSIVE_SEND ( segment_id_local
                   , offset_local
                   , rank
                   , size
                   , timeout )
```

Parameter:

(in) segment_id_local: the local segment ID from which the data is sent

(in) offset_local: the local offset from which the data is sent

(in) rank: the remote rank to which the data is sent

(in) size: the size of the data to be sent

(in) timeout: the timeout

```
gaspi_return_t
gaspi_passive_send ( gaspi_segment_id_t segment_id_local
                   , gaspi_offset_t offset_local
                   , gaspi_rank_t rank
                   , gaspi_size_t size
                   , gaspi_timeout_t timeout )
```

```
function gaspi_passive_send(segment_id_local,offset_local, &
&         rank,size,timeout_ms) &
&         result( res ) bind(C, name="gaspi_passive_send")
  integer(gaspi_segment_id_t), value :: segment_id_local
  integer(gaspi_offset_t), value :: offset_local
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_size_t), value :: size
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_passive_send
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_passive_send` posts a passive communication request which transfers a contiguous block of *size* bytes from a source location of the local GASPI process to the remote GASPI process with the indicated rank *rank*. On the remote side, a corresponding `gaspi_passive_receive` has to be posted. The source location is specified by the pair *segment_id_local*, *offset_local*.

There is a size limit for the data sent with `gaspi_passive_send`. The maximum size is returned by the function `gaspi_passive_transfer_size_max`.

A valid `gaspi_passive_send` communication request requires that the local and the remote segment are allocated and that there is a connection between the local and the remote GASPI process. Otherwise, the communication request is invalid and the procedure returns with `GASPI_ERROR`.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the passive communication request has been posted to the underlying network infrastructure and was completed.

`gaspi_passive_send` calls may be posted from every thread of the GASPI process.

If the procedure returns with `GASPI_TIMEOUT`, the communication request could not be posted to the hardware during the given timeout.

If the passive communication queue is full at the time when a new passive communication request is posted, i. e. the number of posted communication requests has already reached the queue size, the communication request fails and the procedure returns with return value `GASPI_ERROR`.

> *User advice:* Since the passive receive will try to match every corresponding send, the buffer sizes for send/recv need to match for all ranks for the passive communication within one passive send/recv communication step. ⌟

> *User advice:[see also the advice in 8.2.1 on page 61]* It is allowed to write data to the source location while the communication is ongoing. However, the result on the remote side would be some undefined interleaving of the data that was present when the call was issued and the data that was written later.
> It is also allowed to read from the source location while the communcation is ongoing and such a read would retrieve the data written by the application. ⌟

> *User advice:* If the parameter *build_infrastructure* is not set, a connection has to be established between the processes before the `gaspi_passive_send` can be be used. This is accomplished calling the procedure `gaspi_connect`. ⌟

9.2.2 gaspi_passive_receive

The *synchronous non-local time-based blocking* `gaspi_passive_receive` is one of the routines called by the receiver side to engage in passive communication.

```
GASPI_PASSIVE_RECEIVE ( segment_id_local
                      , offset_local
                      , rank
                      , size
                      , timeout )
```

Parameter:

(in) segment_id_local: the local segment ID where to write the data

(in) offset_local: the local offset where to write the data

(out) rank: the remote rank from which the data is transferred

(in) size: the size of the data to be received

(in) timeout: the timeout

```
gaspi_return_t
gaspi_passive_receive ( gaspi_segment_id_t segment_id_local
                      , gaspi_offset_t offset_local
                      , gaspi_rank_t const *rank
                      , gaspi_size_t size
                      , gaspi_timeout_t timeout )
```

```
function gaspi_passive_receive(segment_id_local,offset_local, &
&         rem_rank,size,timeout_ms) &
&         result( res ) bind(C, name="gaspi_passive_receive")
  integer(gaspi_segment_id_t), value :: segment_id_local
  integer(gaspi_offset_t), value :: offset_local
  integer(gaspi_rank_t) :: rem_rank
  integer(gaspi_size_t), value :: size
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_passive_receive
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

gaspi_passive_receive receives a contiguous block of data into a target location from some unspecified remote GASPI process. The target location is specified by the pair *segment_id_local*, *offset_local*.

There is no need for the gaspi_passive_receive procedure to be active before a corresponding gaspi_passive_send procedure is invoked. However, as long as there is no matching receive, the gaspi_passive_send cannot achieve any progress and thus cannot return GASPI_SUCCESS.

The target location needs to have enough space to hold the maximum passive transfer size that could be sent be any other process. Otherwise, the received data might overwrite memory regions outside of the allocated memory and the application will be in an undefined state.

A valid gaspi_passive_receive communication request requires that the local destination segment is allocated and that there is a connection between the local and the remote GASPI process from which a data transfer originates. Otherwise, the communication request is invalid and the procedure returns with GASPI_ERROR.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the data has been received and is available at the target location. Further *rank* contains the rank of the sending process and associated to the communication request.

Successive `gaspi_passive_receive` calls posted by two different threads using two different target locations are allowed. However, the first incoming data is received either by the first thread or the by the second. That means that the `gaspi_passive_receive` should be posted only from a single thread of a GASPI process.

If the procedure returns with `GASPI_TIMEOUT`, there was no pending communication request in the queue. The output parameter *rank* has no defined value.

> *User advice:* It is allowed to write data to the local target location while the passive communication is ongoing. However, the content of the memory would be some undefined interleaving of the data transferred from remote side and the data written locally.
> Also, it is allowed to read from the local target location while the passive communication is ongoing. Such a read would retrieve some undefined interleaving of the data that was present when the call was issued and the data that was transferred from the remote side. ⌟

> *Implementor advice:* A quality implementation enforces fairness in communication that is, no sender should see its communication request delayed indefinitely. The passive keyword means the communication calls shall avoid busy-waiting and consume no CPU cycles, freeing the system for computation. ⌟

9.3 Passive communication utilities

9.3.1 gaspi_passive_queue_purge

The `gaspi_passive_queue_purge` procedure is a *synchronous local time-based blocking* procedure which purges the passive queue.

```
GASPI_PASSIVE_QUEUE_PURGE (timeout)
```

Parameter:

(in) timeout: the timeout

```
gaspi_return_t
gaspi_passive_queue_purge (gaspi_timeout_t timeout)
```

```
function gaspi_passive_queue_purge(timeout) &
&         result( res ) bind(C, name="gaspi_passive_queue_purge")
  integer(gaspi_timeout_t), value :: timeout
  integer(gaspi_return_t) :: res
end function gaspi_passive_queue_purge
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error

This procedure should only be invoked in the situation in which a node failure is detected by inspecting the global health state with gaspi_state_vec_get.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the passive communication queue is purged.

If the procedure returns with GASPI_TIMEOUT, the purge request could not be completed during the given timeout. A subsequent call of gaspi_passive_queue_purge has to be invoked in order to complete the call.

If the procedure returns with GASPI_ERROR, the purge request was not satisfied and returned abnormally.

10 Global atomics

10.1 Introduction and Overview

An atomic operation is an operation which is guaranteed to be executed without fear of interference from other processes during the procedure call. Only one GASPI process at a time has access to the global variable and can modify it.

Atomic operations are also guaranteed to be fair. That means no GASPI process should see its atomic operation request delayed indefinitely.

10.2 Atomic operation calls

10.2.1 gaspi_atomic_fetch_add

The gaspi_atomic_fetch_add procedure is a *synchronous non-local time-based blocking* procedure which atomically adds a given value to a globally acessible value.

```
GASPI_ATOMIC_FETCH_ADD ( segment_id
                       , offset
                       , rank
                       , value_add
                       , value_old
                       , timeout )
```

Parameter:

(in) segment_id: the segment ID where the value is located

(in) offset: the offset where the value is located

(in) rank: the rank where the value is located

(in) value_add: the value which is to be added

(out) value_old: the old value before the operation

(in) timeout: the timeout

```
gaspi_return_t
gaspi_atomic_fetch_add ( gaspi_segment_id_t segment_id
                       , gaspi_offset_t offset
                       , gaspi_rank_t rank
                       , gaspi_atomic_value_t value_add
                       , gaspi_atomic_value_t *value_old
                       , gaspi_timeout_t timeout )
```

```
function gaspi_atomic_fetch_add(segment_id,offset,rank, &
&         val_add,val_old,timeout_ms) &
&         result( res ) bind(C, name="gaspi_atomic_fetch_add")
  integer(gaspi_segment_id_t), value :: segment_id
  integer(gaspi_offset_t), value :: offset
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_atomic_value_t), value :: val_add
  integer(gaspi_atomic_value_t) :: val_old
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_atomic_fetch_add
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_atomic_fetch_add` atomically adds the value of *value_add* to the value on rank *rank*, segment *segment_id_remote* and offset *offset_remote*.

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the parameter *value_old* contains the value before the operation has been applied.

If the procedure returns with `GASPI_TIMEOUT`, the fetch and add request could not be completed during the given timeout. The parameter *value_old* has an undefined value. A subsequent call of `gaspi_atomic_fetch_add` needs to be invoked in order to complete the operation.

If the procedure returns with `GASPI_ERROR`, the fetch and add request aborted abnormally. The parameter *value_ old* as well as the global value (segment_id, offset, rank) have undefined values.

In both cases, `GASPI_TIMEOUT` and `GASPI_ERROR`, the GASPI state vector should be checked in order to deal with possible failures.

> *Implementor advice:* The implementation might require some alignment restrictions that is, the triple(*segment_ id*, *offset*, *rank*) might be required to respect some alignment restrictions. ⌟

> *User advice:* Concurrent accesses to the location represented by the triple(*segment_ id*, *offset*, *rank*) are possible but consistency must be handled by the application. ⌟

10.2.2 gaspi_atomic_compare_swap

The `gaspi_atomic_compare_swap` procedure is a *synchronous non-local time-based blocking* procedure which atomically compares the value of a global value against some user given value and in case these are equal the old value is replaced by a new value.

```
GASPI_ATOMIC_COMPARE_SWAP ( segment_id
                          , offset
                          , rank
                          , comparator
                          , value_new
                          , value_old
                          , timeout )
```

Parameter:

(in) segment_ id: the segment ID where the value is located

(in) offset: the offset where the value is located

(in) rank: the rank where the value is located

(in) comparator: the value which is compared to the remote value

(in) value_ new: the new value to which the remote location is set if the result of the comparison is true

(out) value_ old: the value before the operation

(in) timeout: the timeout

```
gaspi_return_t
gaspi_atomic_compare_swap ( gaspi_segment_id_t segment_id
                          , gaspi_offset_t offset
                          , gaspi_rank_t rank
                          , gaspi_atomic_value_t comparator
                          , gaspi_atomic_value_t value_new
                          , gaspi_atomic_value_t *value_old
                          , gaspi_timeout_t timeout )
```

```
function gaspi_atomic_compare_swap(segment_id,offset,rank,&
&         comparator,val_new,val_old,timeout_ms) &
&         result( res ) bind(C, name="gaspi_atomic_compare_swap")
  integer(gaspi_segment_id_t), value :: segment_id
  integer(gaspi_offset_t), value :: offset
  integer(gaspi_rank_t), value :: rank
  integer(gaspi_atomic_value_t), value :: comparator
  integer(gaspi_atomic_value_t), value :: val_new
  integer(gaspi_atomic_value_t) :: val_old
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_atomic_compare_swap
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_TIMEOUT: operation has run into a timeout

GASPI_ERROR: operation has finished with an error ⌟

gaspi_atomic_compare_swap atomically compares the global value of the the value on rank *rank*, segment *segment_id_remote* and offset *offset_remote* to the value of *comparator*. If the comparison is true, this global value is set to *value_new*. If the comparison is false, it keeps its value.

After successful procedure completion, i. e. return value GASPI_SUCCESS, the parameter *value_old* contains the previous value before the comparison was done.

If the procedure returns with GASPI_TIMEOUT, the compare and swap request could not be completed during the given timeout. The parameter *value_old* has an undefined value. A subsequent call of gaspi_atomic_compare_swap needs to be invoked in order to complete the operation.

If the procedure returns with GASPI_ERROR, the compare and swap request aborted abnormally. The parameter *value_old* as well as well as the global value (segment_id, offset, rank) have undefined values.

In both cases, GASPI_TIMEOUT and GASPI_ERROR, the GASPI state vector should be checked in order to deal with possible failures.

Implementor advice: The implementation might require some alignment restrictions that is, the triple(*segment_id*, *offset*, *rank*) might be required to respect some alignment restrictions. ⌟

User advice: Concurrent accesses to the location represented by the triple(*segment_id*, *offset*, *rank*) are possible but consistency must be handled by the application. ⌟

10.2.3 Examples

The example in listing 15 illustrates the usage of global atomic operations for implementing a global resource lock. The example is implemented with timeout.

Listing 15: GASPI global resource lock implemented with atomic counters

```
#include <GASPI.h>
#include <assert.h>

#define SUCCESS_OR_RETURN(f)                        \
  {                                                 \
    const int ec = (f);                             \
                                                    \
    if (ec != GASPI_SUCCESS)                        \
      {                                             \
        return ec;                                  \
      }                                             \
  }

#define VAL_UNLOCKED 9999999

gaspi_return_t
global_lock_init ( const gaspi_segment_id_t seg,
                   const gaspi_offset_t off,
                   const gaspi_rank_t rank_loc,
                 , const gaspi_timeout_t timeout
                 )
{
  gaspi_rank_t iProc;

  SUCCESS_OR_RETURN (gaspi_proc_rank (&iProc));

  if( iProc == rank_loc)
    {
      gaspi_pointer_t vptr;
      gaspi_atomic_value_t *lock_ptr;

      SUCCESS_OR_RETURN(gaspi_segment_ptr, &vptr);
      lock_ptr = (gaspi_atomic_value_t *) vptr;

      *lock_ptr = VAL_UNLOCKED;
```

```
    }

  SUCCESS_OR_RETURN (gaspi_barrier ( GASPI_GROUP_ALL
                                   , timeout
                                   )
                    );

  return GASPI_SUCCESS;
}

gaspi_return_t
global_try_lock ( const gaspi_segment_id_t seg,
                  const gaspi_offset_t off,
                  const gaspi_rank_t rank_loc,
                , const gaspi_timeout_t timeout
                )
{
  gaspi_rank_t iProc;

  SUCCESS_OR_RETURN (gaspi_proc_rank (&iProc));

  gaspi_atomic_value_t old_value;

  SUCCESS_OR_RETURN (gaspi_atomic_compare_swap ( seg
                                               , off
                                               , rank_loc
                                               , VAL_UNLOCKED
                                               , iProc
                                               , &old_value
                                               , timeout
                                               )
                    );

  return (old_value == VALUE_UNLOCKED) ? GASPI_SUCCESS
                                       : GASPI_ERROR
                                       ;
}

gaspi_return_t
global_unlock ( const gaspi_segment_id_t seg,
                const gaspi_offset_t off,
                const gaspi_rank_t rank_loc,
              , const gaspi_timeout_t timeout
              )
{
  gaspi_rank_t iProc;

  SUCCESS_OR_RETURN (gaspi_proc_rank (&iProc));

  gaspi_atomic_value_t current_value;
```

```
  SUCCESS_OR_RETURN (gaspi_atomic_compare_swap ( seg
                                                , off
                                                , rank_loc
                                                , iProc
                                                , VAL_UNLOCKED
                                                , &current_value
                                                , timeout
                                               )
                    );

  return GASPI_SUCCESS;
}
```

11 Collective communication

11.1 Introduction and overview

Collective operations are collective with respect to a given group. A necessary condition for successful collective procedure completion is that all GASPI processes forming the given group have invoked the operation.

Collective operations support both synchronous and asynchronous implementations as well as time-based blocking. That means, progress towards successful procedure completion can be achieved either inside the call (for a synchronous implementation) or outside of the call (for an asynchronous implementation) before the procedure exits. In the case of a timeout (which is indicated by return value `GASPI_TIMEOUT`) the operation is then continued in the next call of the procedure. This implies that a collective operation may involve several procedure calls until completion. Completion is indicated by return value `GASPI_SUCCESS`.

Collective operations are exclusive per group, i. e. only one collective operation of a specific type on a given group can run at a given time. Starting a specific collective operation before another one of the same kind is not finished on all processes of the group (and marked as such) is not allowed and yields undefined behavior. For example, two allreduce operations for one group can not run at the same time; however, an allreduce and a barrier operation can run at the same time.

The timeout is a necessary condition in order to be able to write failure tolerant code. Timeout = 0 makes an atomic portion of progress in the operation if possible. If progress is possible, the procedure returns as soon as some progress is achieved. Otherwise, the procedure returns immediately. Here, an atomic portion of progress is defined as the smallest set of non-dividable instructions in the current state of the collective operation.

Reduction operations can be defined by the application via callback functions.

User advice: Not every collective operation will be implementable in an asynchronous fashion – for example if a user-defined callback function is used within a global reduction. Progress in this case can only be achieved inside of the call. Especially for large systems this implies that a collective potentially has to be called a substantial number of times in order to complete – especially if used in combination with `GASPI_TEST`. In this combination the called collective immediately returns (after completing local work) and never waits for data from remote processes. A corresponding code fragment in this case would assume the form:

```
while ( (ret = gaspi_allreduce_user(buffer_send
                                  , buffer_receive
                                  , char num
                                  , size_element
                                  , reduce_operation
                                  , reduce_state
                                  , group
                                  , GASPI_TEST)) == GASPI_TIMEOUT)
{
work_on_something_else();
}

if( ret != GASPI_SUCCESS)
{
     handle_error(ret);
}
```

⌟

11.2 Barrier synchronisation

11.2.1 gaspi_barrier

The `gaspi_barrier` procedure is a *collective time-based blocking* procedure. An implementation is free to provide it as a synchronous or an asynchronous procedure.

```
GASPI_BARRIER ( group
              , timeout )
```

Parameter:

(in) group: the group of ranks which should participate in the barrier

(in) timeout: the timeout

```
gaspi_return_t
gaspi_barrier ( gaspi_group_t group
              , gaspi_timeout_t timeout )
```

```
function gaspi_barrier(group,timeout_ms) &
&         result( res ) bind(C, name="gaspi_barrier")
  integer(gaspi_group_t), value :: group
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_barrier
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error

`gaspi_barrier` blocks the caller until all group members of *group* have invoked the procedure or if *timeout* milliseconds have been reached since procedure invocation. After successful procedure completion, i. e. return value `GASPI_SUCCESS`, all group members have invoked the procedure. In case of `GASPI_TIMEOUT` it is unknown whether or not all GASPI processes forming the given group have invoked the call.

Progress towards successful `gaspi_barrier` completion may be achieved even if the procedure exits due to timeout. The barrier is continued in the next call of the procedure. This implies that a barrier operation may involve several `gaspi_barrier` calls until completion.

Barrier operations are exclusive per group, i. e. only one barrier operation on a given group can run at a time. Starting a barrier operation in another thread before a previously invoked barrier is finished on all processes of the group is not allowed and yields undefined behavior.

In case of error, the return value is `GASPI_ERROR`. The error vector should be investigated.

> *User advice:* The barrier is supposed to synchronise processes and not threads.

11.2.2 Examples

In the following example a `gaspi_barrier` is interrupted after 100 ms in order to check for errors.

```
gaspi_return_t err;

do
  {
    err = gaspi_barrier (g, 100);
```

```

    if (err == GASPI_TIMEOUT && error vector indicates error)
      {
        goto ERROR_HANDLING;
      }
  }
while (err != GASPI_SUCCESS);
```

The following example shows a non-blocking barrier. Some local work (in this case: cleanup) is performed, overlapping it with the barrier and only then a full synchronisation is achieved by calling the barrier again with a blocking semantics (if needed).

```
const gaspi_return_t err = gaspi_barrier (g, GASPI_TEST);

do_local_cleanup();

if (err != GASPI_ERROR && err != GASPI_SUCCESS)
{
    gaspi_barrier (g, GASPI_BLOCK);
}
```

11.3 Predefined global reduction operations

11.3.1 gaspi_allreduce

The gaspi_allreduce procedure is a *collective time-based blocking* procedure. An implementation is free to provide it as a synchronous or an asynchronous procedure.

```
GASPI_ALLREDUCE ( buffer_send
                , buffer_receive
                , num
                , operation
                , datatype
                , group
                , timeout )
```

Parameter:

(in) buffer_send: pointer to the buffer where the input is placed

(in) buffer_receive: pointer to the buffer where the result is placed

(in) num: the number of elements to be reduced on each process

(in) operation: the GASPI reduction operation type

(in) datatype: the GASPI element type

(in) group: the group of ranks which participate in the reduction operation

(in) timeout: the timeout

```
gaspi_return_t
gaspi_allreduce ( gaspi_const_pointer_t buffer_send
                , gaspi_pointer_t buffer_receive
                , gaspi_number_t num
                , gaspi_operation_t operation
                , gaspi_datatype_t datatype
                , gaspi_group_t group
                , gaspi_timeout_t timeout )
```

```
function gaspi_allreduce(buffer_send,buffer_receive,num, &
&         operation,datatyp,group,timeout_ms) &
&         result( res ) bind(C, name="gaspi_allreduce")
  type(c_ptr), value :: buffer_send
  type(c_ptr), value :: buffer_receive
  integer(gaspi_number_t), value :: num
  integer(gaspi_int), value :: operation
  integer(gaspi_int), value :: datatyp
  integer(gaspi_group_t), value :: group
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_allreduce
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_allreduce` combines the *num* elements of type *datatype* residing in *buffer_send* on each process in accordance with the given *operation*. The reduction operation is on a per element basis, i. e. the operation is applied to each of the elements. `gaspi_allreduce` blocks the caller until all data is available that is needed to calculate the result or if *timeout* milliseconds have been reached since procedure invocation. After successful procedure completion, i. e. return value `GASPI_SUCCESS`, all group members have invoked the procedure and *buffer_receive* contains the result of the reduction operation on every GASPI process of *group*. In case of `GASPI_TIMEOUT` not all data is available that is needed to calculate the result.

Progress towards successful `gaspi_allreduce` completion may be achieved even if the procedure exits due to timeout. The reduction operation is continued in the next call of the procedure. This implies that a reduction operation may involve several `gaspi_allreduce` calls until completion.

Reduction operations are exclusive per group, i. e. only one reduction operation on a given group can run at a time. Starting a reduction operation for the same group in a separate thread before previously invoked operation is finished on all processes of the group is not allowed and yields undefined behavior.

The *buffer_send* as well as the *buffer_receive* do not need to reside in the global address space. `gaspi_allreduce` copies the send buffer into an internal buffer at the first invocation. The result is copied from an internal buffer into the receive buffer immediatley before the procedure returns successfully. The buffers need to have the appropriate size to host all of the *num* elements. Otherwise the reduction operation yields undefined behavior. The maximum permissible number of elements is implementation dependent and can be retrieved by `gaspi_allreduce_elem_max`.

In case of error, the return value is `GASPI_ERROR`. The error vector should be examined. *buffer_receive* has an undefined value.

In case of `GASPI_TIMEOUT`, the reduction operation is not finished yet, i. e. not all data is available that is needed to calculate the result. The *buffer_receive* has an undefined value.

11.3.2 Predefined reduction operations

There are three predefined reduction operations:

```
typedef enum { GASPI_OP_MIN
             , GASPI_OP_MAX
             , GASPI_OP_SUM
             } gaspi_operation_t;
```

GASPI_OP_MIN determines the minimum of the elements of each column of the input vector.

GASPI_OP_MAX determines the maximum of the elements of each column of the input vector.

GASPI_OP_SUM sums up all elements of each column of the input vector.

11.3.3 Predefined types

And the types are:

```
typedef enum { GASPI_TYPE_INT
             , GASPI_TYPE_UINT
             , GASPI_TYPE_LONG
             , GASPI_TYPE_ULONG
             , GASPI_TYPE_FLOAT
             , GASPI_TYPE_DOUBLE
             } gaspi_datatype_t;
```

GASPI_TYPE_INT integer

GASPI_TYPE_UINT unsigned integer

GASPI_TYPE_LONG long

GASPI_TYPE_ULONG unsigned long

GASPI_TYPE_FLOAT float

GASPI_TYPE_DOUBLE double

11.4 User-defined global reduction operations

11.4.1 `gaspi_allreduce_user`

The procedure `gaspi_allreduce_user` allows the user to specify its own reduction operation. Only operations are supported which are commutative and associative. It is a *collective time-based blocking* procedure. An implementation is free to provide it as a synchronous or an asynchronous procedure.

```
GASPI_ALLREDUCE_USER ( buffer_send
                     , buffer_receive
                     , num
                     , size_element
                     , reduce_operation
                     , reduce_state
                     , group
                     , timeout )
```

Parameter:

(in) buffer_send: pointer to the buffer where the input is placed

(in) buffer_receive: pointer to the buffer where the result is placed

(in) num: the number of elements to be reduced on each process

(in) size_element: Size in bytes of one element to be reduced

(in) reduce_operation: pointer to the user defined reduction operation procedure

(inout) reduce_state: reduction state vector

(in) group: the group of ranks which participate in the reduction operation

(in) timeout: the timeout

```
gaspi_return_t
gaspi_allreduce_user ( gaspi_const_pointer_t buffer_send
                     , gaspi_pointer_t buffer_receive
                     , gaspi_number_t num
                     , gaspi_size_t size_element
                     , gaspi_reduce_operation_t reduce_operation
                     , gaspi_reduce_state_t reduce_state
                     , gaspi_group_t group
                     , gaspi_timeout_t timeout )
```

```
function gaspi_allreduce_user(buffer_send,buffer_receive, &
&         num,element_size,reduce_operation,reduce_state,&
&         group,timeout_ms) &
&         result( res ) bind(C, name="gaspi_allreduce_user")
  type(c_ptr), value :: buffer_send
  type(c_ptr), value :: buffer_receive
  integer(gaspi_number_t), value :: num
  integer(gaspi_size_t), value :: element_size
  type(c_funptr), value :: reduce_operation
  type(c_ptr), value :: reduce_state
  integer(gaspi_group_t), value :: group
  integer(gaspi_timeout_t), value :: timeout_ms
  integer(gaspi_return_t) :: res
end function gaspi_allreduce_user
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error ⌟

`gaspi_allreduce_user` has the same semantics as the predefined reduction operation `gaspi_allreduce` described in the last section.

A user defined reduction operation *reduce_ operation* and a user defined state *reduce_ state* are passed.

The elements on which the user defined reduction operation is applied are described by their byte size *size_ element*. The entire size of the data to be reduced, i. e. *num* times *size_ element*, must not be larger than the internal buffer size of `gaspi_allreduce_user`. The internal buffer size can be queried through `gaspi_allreduce_buf_size`.

11.4.2 User defined reduction operations

The prototype for the user defined reduction operations is the following:

```
GASPI_REDUCE_OPERATION ( operand_one
                       , operand_two
                       , result
                       , state
                       , timeout )
```

Parameter:

(in) operand_one: pointer to the first operand

(in) operand_two: pointer to the second operand

(in) result: pointer to the result

(in) state: pointer to the state

(in) timeout: the timeout

```
gaspi_return_t
gaspi_reduce_operation ( gaspi_const_pointer_t operand_one
                       , gaspi_const_pointer_t operand_two
                       , gaspi_pointer_t result
                       , gaspi_reduce_state_t state
                       , gaspi_timeout_t timeout )
```

```
function my_reduce_operation(op_one,op_two,op_res, &
&   op_state,num,element_size,timeout) &
&   result ( res ) bind(C,name="my_reduce_operation")
  implicit none
  integer(gaspi_number_t), intent(in), value :: num
!
! the fortran user defined callback function requires an
! explicit type from the iso_c_binding module. in this
! example integer(c_int) (op_one,op_two,op_res,op_state)
!
  integer(c_int), intent(in)  :: op_one(num)
  integer(c_int), intent(in)  :: op_two(num)
  integer(c_int), intent(out) :: op_res(num)
  integer(c_int), intent(out) :: op_state(num)
  integer(gaspi_size_t), value :: element_size
  integer(gaspi_timeout_t), value :: timeout
  integer(gaspi_return_t) :: res
!
! your user defined operation
! ...
  res = GASPI_SUCCESS
end function my_reduce_operation
```

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_TIMEOUT`: operation has run into a timeout

`GASPI_ERROR`: operation has finished with an error

A pointer to the first operand and a pointer to the second operand are passed. The result is stored in the memory represented by the pointer *result.* In addition to the actual data, also a state can be passed to the operator which might be required in order to compute the result. In order to meet real time system specifications, a timeout can be passed to the user defined reduction operator. The reduction operator should return a `gaspi_return_t` with the same semantics, i. e. `GASPI_SUCCESS` for successful procedure completion. `GASPI_TIMEOUT` in case of timeout and `GASPI_ERROR` in case of error.

The user defined reduction operator needs to be commutative and associative.

The reduce operator type passed to `gaspi_allreduce_user` is a pointer to a function with the prototype described above.

typedef `gaspi_reduce_operation`* `gaspi_reduce_operation_t`

The GASPI *reduction operation type*

11.4.3 allreduce state

The allreduce state type

typedef void* `gaspi_reduce_state_t`

The GASPI *reduction operation state type*

is a pointer to a state which may be passed to the user defined reduction operation. A state may contain additional information beside the actual data to be reduced needed to perform the reduction operation.

11.4.4 Example

A fortran version of the user defined allreduce hence might assume the form listing 16

Listing 16: GASPI User defined allreduce, fortran example.

```
module my_reduce

  use gaspi_c_binding
  implicit none

contains

  function my_reduce_operation(op_one,op_two,op_res, &
&     op_state,num,element_size,timeout) &
```

```
&     result ( res ) bind(C,name="my_reduce_operation")
    implicit none
    integer(gaspi_number_t), intent(in), value :: num
    integer(c_int), intent(in) :: op_one(num)
    integer(c_int), intent(in) :: op_two(num)
    integer(c_int), intent(out) :: op_res(num)
    integer(c_int), intent(out) :: op_state(num)
    integer(gaspi_size_t), value :: element_size
    integer(gaspi_timeout_t), value :: timeout
    integer(gaspi_return_t) :: res
    integer i
    do i = 1, num
       op_res(i) = max(op_one(i),op_two(i))
    enddo
    res = GASPI_SUCCESS
  end function my_reduce_operation

end module my_reduce

program allreduce

  use gaspi_c_binding
  use my_reduce
  implicit none
  integer(gaspi_size_t) :: sizeof_int
  integer(gaspi_return_t) :: res
  integer(gaspi_rank_t) :: rank
  integer(c_int), dimension(1), target :: buffer_send
  integer(c_int), dimension(1), target :: buffer_recv
  integer(c_int), dimension(1), target :: reduce_state
  integer(gaspi_number_t) :: num_elem
  integer(gaspi_group_t) :: group
  integer(gaspi_timeout_t) :: timeout
  type(c_funptr) :: fproc

  sizeof_int = 4
  num_elem = 1
  group = GASPI_GROUP_ALL
  timeout = GASPI_BLOCK
  fproc = c_funloc(my_reduce_operation)
  res = gaspi_proc_init(timeout)
  res = gaspi_proc_rank(rank)

  buffer_send(1) = rank
  buffer_recv(1) = -1
  reduce_state(1) = 0
  res = gaspi_allreduce_user(C_LOC(buffer_send),&
&       C_LOC(buffer_recv),num_elem,sizeof_int,&
&       fproc,C_LOC(reduce_state),&
&       group,timeout)
```

```

  res = gaspi_proc_term(timeout)

end program allreduce
```

12 GASPI getter functions

The GASPI specification provides getter functions for all entries in the GASPI configuration. These getter functions are *synchronous local blocking* procedures which, after successful procedure completion (i. e. return value GASPI_SUCCESS), read out the corresponding value of the current configuration setting.

The values of the parameters in the GASPI configuration are determined in gaspi_proc_init at startup. If the value of one of these parameters is compliant with the system capabilities, the parameter is set to the requested/preferred value. Otherwise, the parameter is set to the maximum value compliant with the system capabilities. The values of the parameters realised in the GASPI configuration are implementation specific.

In case of error, the return value is GASPI_ERROR and the corresponding parameter in the getter function has an undefined value.

12.1 Getter functions for group management

12.1.1 gaspi_group_max

```
GASPI_GROUP_MAX (group_max)
```

Parameter:

(out) group_max: the total number of groups

```
gaspi_return_t
gaspi_group_max (gaspi_number_t *group_max)
```

```
function gaspi_group_max(group_max) &
&         result( res ) bind(C, name="gaspi_group_max")
  integer(gaspi_number_t) :: group_max
  integer(gaspi_return_t) :: res
end function gaspi_group_max
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

12.2 Getter functions for segment management

12.2.1 gaspi_segment_max

```
GASPI_SEGMENT_MAX (segment_max)
```

Parameter:

(out) segment_max: the total number of permissible segments

```
gaspi_return_t
gaspi_segment_max (gaspi_number_t *segment_max)
```

```
function gaspi_segment_max(segment_max) &
&         result( res ) bind(C, name="gaspi_segment_max")
  integer(gaspi_number_t) :: segment_max
  integer(gaspi_return_t) :: res
end function gaspi_segment_max
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

12.3 Getter functions for communication management

12.3.1 gaspi_queue_num

```
GASPI_QUEUE_NUM (queue_num)
```

Parameter:

(out) queue_num: the number of available queues

```
gaspi_return_t
gaspi_queue_num (gaspi_number_t *queue_num)
```

```
function gaspi_queue_num(queue_num) &
&         result( res ) bind(C, name="gaspi_queue_num")
  integer(gaspi_number_t) :: queue_num
  integer(gaspi_return_t) :: res
end function gaspi_queue_num
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

12.3.2 gaspi_queue_size_max

```
GASPI_QUEUE_SIZE_MAX ( queue_size_max )
```

Parameter:

(out) queue_size_max: the maximum number of simultaneous requests allowed

```
gaspi_return_t
gaspi_queue_size_max ( gaspi_number_t* queue_size_max )
```

```
function gaspi_queue_size_max(queue_size_max) &
&         result( res ) bind(C, name="gaspi_queue_size_max")
  integer(gaspi_number_t) :: queue_size_max
  integer(gaspi_return_t) :: res
end function gaspi_queue_size_max
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

12.3.3 gaspi_queue_max

```
GASPI_QUEUE_MAX ( queue_max )
```

Parameter:

(out) queue_max: the maximum number of allowed queues

```
gaspi_return_t
gaspi_queue_max ( gaspi_number_t queue_max )
```

```
function gaspi_queue_max ( queue_max ) &
&        result(res) bind (C, name="gaspi_queue_max" )
  integer(gaspi_number_t), value :: queue_max
  integer(gaspi_return_t) :: res
end function gaspi_queue_max
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

12.3.4 `gaspi_transfer_size_max`

```
GASPI_TRANSFER_SIZE_MAX (transfer_size_max)
```

Parameter:

(out) transfer_size_max: the maximum transfer size allowed for a single request

```
gaspi_return_t
gaspi_transfer_size_max (gaspi_size_t *transfer_size_max)
```

```
function gaspi_transfer_size_max(transfer_size_max) &
&         result( res ) &
&         bind(C, name="gaspi_transfer_size_max")
  integer(gaspi_size_t) :: transfer_size_max
  integer(gaspi_return_t) :: res
end function gaspi_transfer_size_max
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

12.3.5 `gaspi_notification_num`

```
GASPI_NOTIFICATION_NUM (notification_num)
```

Parameter:

(out) notification_num: the number of available notifications

```
gaspi_return_t
gaspi_notification_num (gaspi_number_t *notification_num)
```

```
function gaspi_notification_num(notification_num) &
&         result( res ) bind(C, name="gaspi_notification_num")
  integer(gaspi_number_t) :: notification_num
  integer(gaspi_return_t) :: res
end function gaspi_notification_num
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

12.4 Getter functions for passive communication

12.4.1 gaspi_passive_transfer_size_max

```
GASPI_PASSIVE_TRANSFER_SIZE_MAX (transfer_size_max)
```

Parameter:

(out) transfer_size_max: maximal transfer size per single passive communication request

```
gaspi_return_t
gaspi_passive_transfer_size_max (gaspi_size_t *transfer_size_max)
```

```
function gaspi_passive_transfer_size_max(transfer_size_max) &
&         result( res ) &
&         bind(C, name="gaspi_passive_transfer_size_max")
  integer(gaspi_size_t) :: transfer_size_max
  integer(gaspi_return_t) :: res
end function gaspi_passive_transfer_size_max
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

12.5 Getter functions related to atomic operations

12.5.1 gaspi_atomic_max

```
GASPI_ATOMIC_MAX (max_value)
```

Parameter:

(out) max_value: the maximum value an gaspi_atomic_value_t can hold

```
gaspi_return_t
gaspi_atomic_max (gaspi_atomic_value_t *max_value)
```

```
function gaspi_atomic_max(max_value) &
&         result( res ) bind(C, name="gaspi_atomic_max")
  integer(gaspi_atomic_value_t) :: max_value
  integer(gaspi_return_t) :: res
end function gaspi_atomic_max
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

12.6 Getter functions for collective communication

12.6.1 `gaspi_allreduce_buf_size`

```
GASPI_ALLREDUCE_BUF_SIZE (buf_size)
```

Parameter:

(out) buf_size: the size of the internal buffer in `gaspi_allreduce_user`

```
gaspi_return_t
gaspi_allreduce_buf_size (gaspi_size_t *buf_size)
```

```
function gaspi_allreduce_buf_size(buf_size) &
&         result( res ) bind(C, name="gaspi_allreduce_buf_size")
  integer(gaspi_size_t) :: buf_size
  integer(gaspi_return_t) :: res
end function gaspi_allreduce_buf_size
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

12.6.2 gaspi_allreduce_elem_max

```
GASPI_ALLREDUCE_ELEM_MAX (elem_max)
```

Parameter:

(out) elem_max: the maximum number of elements allowed in `gaspi_allreduce`

```
gaspi_return_t
gaspi_allreduce_elem_max (gaspi_number_t *elem_max)
```

```
function gaspi_allreduce_elem_max(elem_max) &
&         result( res ) bind(C, name="gaspi_allreduce_elem_max")
  integer(gaspi_number_t) :: elem_max
  integer(gaspi_return_t) :: res
end function gaspi_allreduce_elem_max
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error

12.7 Getter functions related to infrastructure

12.7.1 gaspi_network_type

```
GASPI_NETWORK_TYPE (network_type)
```

Parameter:

(out) network_type: the chosen network type

```
gaspi_return_t
gaspi_network_type (gaspi_network_t *network_type)
```

```
function gaspi_network_type(network_type) &
&         result( res ) bind(C, name="gaspi_network_type")
  integer(gaspi_network_t) :: network_type
  integer(gaspi_return_t) :: res
end function gaspi_network_type
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

12.7.2 gaspi_build_infrastructure

```
GASPI_BUILD_INFRASTRUCTURE (build_infrastructure)
```

Parameter:

(out) build_infrastructure: the current value of *build_infrastructure*

```
gaspi_return_t
gaspi_build_infrastructure (gaspi_number_t *build_infrastructure)
```

```
function gaspi_build_infrastructure(build_infrastructure) &
&         result( res ) &
&         bind(C, name="gaspi_build_infrastructure")
  integer (gaspi_number_t) :: build_infrastructure
  integer(gaspi_return_t) :: res
end function gaspi_build_infrastructure
```

Execution phase:

Working

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

13 GASPI Environmental Management

13.1 Implementation Information

13.1.1 gaspi_version

The gaspi_version procedure is a *synchronous local blocking* procedure which determines the version of the running GASPI installation.

```
GASPI_VERSION (version)
```

Parameter:

(out) version: The version of the running GASPI installation

```
gaspi_return_t
gaspi_version (float *version)
```

```
function gaspi_version(version) &
&         result( res ) bind(C, name="gaspi_version")
  real(gaspi_float) :: version
  integer(gaspi_return_t) :: res
end function gaspi_version
```

Execution phase:

Any

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error

After successful procedure completion, i. e. return value `GASPI_SUCCESS` *version* contains the version of the running GASPI installation.

In case of error, the return value is `GASPI_ERROR`. The output parameter *version* has an undefined value.

13.2 Timing information

13.2.1 gaspi_time_get

The `gaspi_time_get` procedure is a *synchronous local blocking* procedure which determines the time elapsed since an arbitrary point of time in the past.

```
GASPI_TIME_GET (wtime)
```

Parameter:

(out) wtime: time elapsed in milliseconds

```
gaspi_return_t
gaspi_time_get (gaspi_time_t *wtime)
```

```
function gaspi_time_get(wtime) &
&         result( res ) bind(C, name="gaspi_time_get")
  integer(gaspi_time_t) :: wtime
  integer(gaspi_return_t) :: res
end function gaspi_time_get
```

Execution phase:

Working

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the parameter *wtime* contains elapsed time in milliseconds since an arbitrary point in the past. The parameter *wtime* is not synchronised among the different GASPI processes.

In case of error, the return value is `GASPI_ERROR`. The value of the output parameter *wtime* is undefined.

13.2.2 gaspi_time_ticks

The `gaspi_time_ticks` procedure is a *synchronous local blocking* procedure which returns the resolution of the internal timer in terms of milliseconds.

```
GASPI_TIME_TICKS (resolution)
```

Parameter:

(out) resolution: the resolution of the internal timer in milliseconds

```
gaspi_return_t
gaspi_time_ticks (gaspi_time_t *resolution)
```

```
function gaspi_time_ticks(resolution) &
&         result( res ) bind(C, name="gaspi_time_ticks")
  integer(gaspi_time_t) :: resolution
  integer(gaspi_return_t) :: res
end function gaspi_time_ticks
```

Execution phase:

Any

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the parameter *resolution* contains the resolution of the internal timer in units of milliseconds.

In case of error, the return value is `GASPI_ERROR`. The value of the output parameter *resolution* is undefined.

13.3 Error Codes and Classes

13.3.1 GASPI error codes

In principle all return values less than zero represent an error. Every implementation is free to define specific error codes.

13.3.2 gaspi_print_error

The `gaspi_print_error` procedure is a *synchronous local blocking* procedure which translates an error code to a text message.

```
GASPI_PRINT_ERROR( error_code
                 , error_message )
```

Parameter:

(in) error_code: the error code to be translated

(out) error_message: the error message

```
gaspi_return_t
gaspi_print_error( gaspi_return_t error_code
                 , gaspi_string_t *error_message )
```

```
function gaspi_print_error(error_code,error_message) &
&         result( res ) bind(C, name="gaspi_print_error")
  integer(gaspi_return_t), value :: error_code
  character(c_char), dimension(*) :: error_message
  integer(gaspi_return_t) :: res
end function gaspi_print_error
```

Execution phase:

Any

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error

After successful procedure completion, i. e. return value `GASPI_SUCCESS` *error_message* contains the error message corresponding to the error code *error_code*.

In case of error, the return value is `GASPI_ERROR`.

The procedure can be invoked in any of the GASPI execution phases.

14 Profiling Interface

The profiling interface of GASPI consists of two parts. The statistics part provides the means to allow the user to collect basic profiling data about a program run. The event tracing part describes the requirements for an GASPI implementation in order to support the transparent interception and inspection of function calls.

14.1 Statistics

14.1.1 gaspi_statistic_counter_max

The gaspi_statistic_counter_max procedure is a *synchronous local blocking* procedure, which provides a way to inform the GASPI user dynamically about the number of avialable counters. An implementation should not provide a compile-time constant maximum for gaspi_statistic_counter_t. Instead the user can call gaspi_statistic_counter_max in order to determine the maximum value for gaspi_statistic_counter_t.

```
GASPI_STATISTIC_COUNTER_MAX ( counter_max )
```

Parameter:

(out) counter_max: the maximum value for gaspi_statistic_counter_t. The allowed value range is $0 \leq counter < counter\_max$

```
gaspi_return_t
gaspi_statistic_counter_max ( gaspi_number_t *counter_max )
```

```
function gaspi_statistic_counter_max(counter_max) &
&         result( res ) &
&         bind(C, name="gaspi_statistic_counter_max")
  integer(gaspi_statistic_counter_t) :: counter_max
  integer(gaspi_return_t) :: res
end function gaspi_statistic_counter_max
```

Execution phase:

Any

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

If a GASPI implementation defines symbolic constants for gaspi_statistic_counter_t a priori, then gaspi_statistic_counter_max should set *counter_max* to the corresponding maximum value. A high-speed implementation will likely set *counter_max* to 0 and does not provide any statistics by default. A dynamically linked wrapper library can provide extra counters by adjusting the return value of `gaspi_statistic_counter_max`.

> *Library implementor advice:* A sensible wrapper library will respect the value returned by the native `gaspi_statistic_counter_max` and append their own counters accordingly. Thus accessses to statistic counters provided by the GASPI implementation itself are not harmed. ⌟

14.1.2 `gaspi_statistic_counter_info`

The `gaspi_statistic_counter_info` procedure is a *synchronous local blocking* procedure which provides an implementation independent way to retrieve information for a particular statistic counter. Beside the name and a description this function also yields the meaning of the argument value for this counter, if any. The meaning is defined in terms of the `gaspi_statistic_argument_t` enumeration.

```
typedef enum { GASPI_STATISTIC_ARGUMENT_NONE
             , GASPI_STATISTIC_ARGUMENT_RANK
             , ...
             } gaspi_statistic_argument_t;
```

A GASPI implementation is free to extend the above enumeration.

```
GASPI_STATISTIC_COUNTER_INFO ( const counter
                             , argument
                             , counter_name
                             , counter_description
                             , verbosity_level )
```

Parameter:

(in) counter: the counter, for which detailed information is requested

(out) counter_argument: the meaning of the argument value

(out) counter_name: a short name of this counter

(out) counter_description: a more verbose description of this counter

(out) verbosity_level: minimum verbosity level to activate this counter (at least 1)

```
gaspi_return_t
gaspi_statistic_counter_info ( gaspi_statistic_counter_t counter
                             , gaspi_statistic_argument_t
                                     *argument
                             , gaspi_string_t *counter_name
                             , gaspi_string_t
                                     *counter_description
                             , gaspi_number_t *verbosity_level )
```

```
function gaspi_statistic_counter_info(counter,counter_argument, &
&         counter_name,counter_description,verbosity_level) &
&         result( res ) &
&         bind(C, name="gaspi_statistic_counter_info")
  integer(gaspi_statistic_counter_t), value :: counter
  integer(gaspi_statistic_argument_t) :: counter_argument
  character(c_char), dimension(*) :: counter_name
  character(c_char), dimension(*) :: counter_description
  integer(gaspi_number_t) :: verbosity_level
  integer(gaspi_return_t) :: res
end function gaspi_statistic_counter_info
```

Execution phase:

Any

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error ⌟

After successful procedure completion, i. e. return value `GASPI_SUCCESS`, the out variables contain the desired information. A dynamically linked wrapper library should provide information for added counters by wrapping `gaspi_statistic_counter_info`. The verbosity level for all counters should be at least 1 (see `gaspi_statistic_verbosity_level` below).

If the return value is `GASPI_ERROR`, the particular *counter* issued to `gaspi_statistic_counter_info` does not exist.

14.1.3 `gaspi_statistic_verbosity_level`

The `gaspi_statistic_verbosity_level` procedure is a *synchronous local blocking* procedure which sets the process-wide verbosity level of the statistic interface. A counter is only active (that is, it is updated), if the process-wide verbosity level is higher or equal to the minimum verbosity level of that counter. If a call to `gaspi_statistic_verbosity_level` activates or deactivates counters and there are *asynchronous* operations in progress, it is unspecified, whether and how these counters are affected by the operations. It is furthermore unspecified whether and how counters of higher verbositiy levels are updated.

A verbosity level of 0 deactivates all counting. It is not guaranteed, that counters with a minimum verbosity level of 0 are counted properly, if the verbosity level is set to 0.

```
GASPI_STATISTIC_VERBOSITY_LEVEL ( verbosity_level )
```

Parameter:

(in) verbosity_level: the level of desired verbosity

```
gaspi_return_t
gaspi_statistic_verbosity_level ( gaspi_number_t verbosity_level)
```

```
function gaspi_statistic_verbosity_level(verbosity_level_) &
&         result( res ) &
&         bind(C, name="gaspi_statistic_verbosity_level")
  integer(gaspi_number_t), value :: verbosity_level_
  integer(gaspi_return_t) :: res
end function gaspi_statistic_verbosity_level
```

Execution phase:

Any

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error

14.1.4 gaspi_statistic_counter_get

The `gaspi_statistic_counter_get` procedure is a *synchronous local blocking* procedure which retrieves a statistical counter from the local GASPI process.

```
GASPI_STATISTIC_COUNTER_GET ( counter
                            , argument
                            , value )
```

Parameter:

(in) counter: the counter to be retrieved

(in) argument: the argument for the counter

(out) value: the current value of the counter

```
gaspi_return_t
gaspi_statistic_counter_get ( gaspi_statistic_counter_t counter
                            , gaspi_statistic_argument_t argument
                            , gaspi_number_t *value )
```

```
function gaspi_statistic_counter_get(counter,argument,&
&         value_arg) &
&         result( res ) &
&         bind(C, name="gaspi_statistic_counter_get")
  integer(gaspi_statistic_counter_t), value :: counter
  integer(gaspi_statistic_argument_t), value :: argument
  integer(gaspi_number_t) :: value_arg
  integer(gaspi_return_t) :: res
end function gaspi_statistic_counter_get
```

Execution phase:

Any

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

The meaning of parameter *argument* depends on the retrieved counter. For instance, if a counter retrieves the bytes sent per target rank, then *argument* contains the target rank number. If the retrieved counter has no argument, the value of *argument* is ignored. After successful procedure completion, i. e. return value GASPI_SUCCESS *value* contains the current value of the corresponding counter.

The return value is GASPI_ERROR, if *counter* does not exist, i.e. exceeds gaspi_statistic_counter_max.

It is allowed to access a counter even, if the process-wide verbosity level is lower than the minimum verbosity level of that counter. Thus it is possible to profile certain regions of an application by changing the verbosity level and read the counter values at a later point in time independently of the current verbosity level.

14.1.5 gaspi_statistic_counter_reset

The gaspi_statistic_counter_reset procedure is a *synchronous local blocking* procedure which sets a statistical counter to 0.

```
GASPI_STATISTIC_COUNTER_RESET (counter)
```

Parameter:

(in) counter: the counter to be reset

```
gaspi_return_t
gaspi_statistic_counter_reset (gaspi_statistic_counter_t counter)
```

```
function gaspi_statistic_counter_reset(counter) &
&         result( res ) &
&         bind(C, name="gaspi_statistic_counter_reset")
  integer(gaspi_statistic_counter_t), value :: counter
  integer(gaspi_return_t) :: res
end function gaspi_statistic_counter_reset
```

Execution phase:

Any

Return values:

`GASPI_SUCCESS`: operation has returned successfully

`GASPI_ERROR`: operation has finished with an error

The return value is `GASPI_ERROR`, if *counter* does not exist, i.e. exceeds `gaspi_statistic_counter_max`.

14.2 Event Tracing

The GASPI event tracing interface defines the requirements for an implementation to support the transparent interception and inspection of GASPI calls. A GASPI implementation must provide a mechanism, through which all GASPI functions may be accessed with a name shift. The alternate entry point names have the prefix `pgaspi_` instead of `gaspi_`. In addition the function `gaspi_pcontrol` is provided.

14.2.1 gaspi_pcontrol

The function `gaspi_pcontrol` is a no-op. A GASPI implementation itself ignores the value of *argument* and returns immediately.

This routine is provided in order to enable users to communicate with an event trace interface from inside the application. The meaning of *argument* is specified by the used event tracer.

```
GASPI_PCONTROL ( argument )
```

Parameter:

(inout) argument:

```
gaspi_return_t
gaspi_pcontrol ( gaspi_pointer_t argument )
```

```
function gaspi_pcontrol(argument) &
&         result( res ) bind(C, name="gaspi_pcontrol")
  type(c_ptr), value :: argument
  integer(gaspi_return_t) :: res
end function gaspi_pcontrol
```

Execution phase:

Any

Return values:

GASPI_SUCCESS: operation has returned successfully

GASPI_ERROR: operation has finished with an error

A Listings

A.1 success_or_die

Listing 17: success_or_die.h

```
#ifndef _SUCCESS_OR_DIE_H
#define _SUCCESS_OR_DIE_H 1

void success_or_die ( const char* file, const int line
                    , const int ec
                    );

#ifndef NDEBUG
#define ASSERT(ec) success_or_die (__FILE__, __LINE__, ec)
#else
#define ASSERT(ec) ec
#endif

#endif
```

Listing 18: success_or_die.c

```
#include <success_or_die.h>
#include <stdlib.h>
#include <stdio.h>
#include <GASPI.h>

void success_or_die ( const char* file, const int line
                    , const int ec
                    )
{
  if (ec != GASPI_SUCCESS)
    {
```

```
    gaspi_string_t str;

    gaspi_error_message (ec, &str);

    fprintf (stderr, "error in %s[%i]: %s\n", file, line, str);

    exit (EXIT_FAILURE);
  }
}
```

A.2 wait_if_queue_full

Listing 19: wait_if_queue_full.h

```
#ifndef _WAIT_IF_QUEUE_FULL_H
#define _WAIT_IF_QUEUE_FULL_H 1

#include <GASPI.h>

void wait_if_queue_full ( const gaspi_queue_id_t queue_id
                        , const gaspi_number_t request_size
                        );

#endif
```

Listing 20: wait_if_queue_full.c

```
#include <wait_if_queue_full.h>
#include <success_or_die.h>

void wait_if_queue_full ( const gaspi_queue_id_t queue_id
                        , const gaspi_number_t request_size
                        )
{
  gaspi_number_t queue_size_max;
  gaspi_number_t queue_size;

  ASSERT (gaspi_queue_size_max (&queue_size_max));
  ASSERT (gaspi_queue_size (queue_id, &queue_size));

  if (queue_size + request_size >= queue_size_max)
    {
      ASSERT (gaspi_wait (queue_id, GASPI_BLOCK));
    }
}
```