Errata proposal - GASPI_ERROR_END_OF_QUEUE

Christian Simmendinger

December 14, 2016

1 GASPI_QUEUE_FULL

In order to resolve problems in portability and in order to provide improved handling of race conditions, we suggest that the queue related function calls gaspi_notify, gaspi_write, gaspi_write_notify, gaspi_write_list_notify, gaspi_read, gaspi_read_notify and gaspi_read_list_notify return a predefined return value when reaching the end of the respective queue, namely GASPI_QUEUE_FULL.

2 Needed Resources

• none.

3 Additional (necessary) Changes to the Standard

3.1 Return values

Gaspi procedures have four 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. The function gaspi_error_message translates the error code into a human readable format.

GASPI_QUEUE_FULL implies that one of the function calls gaspi_notify, gaspi_write, gaspi_write_notify, gaspi_write_list_notify, gaspi_read, gaspi_read_notify and gaspi_read_list_notify has reached the end of the
queue and that the corresponding communication request could not be issued. If GASPI_QUEUE_FULL is returned, users should either switch to another queue or wait (see gaspi_wait) and subsequently re-issue the communication request.

3.2 Constants

| GASPI_QUEUE_FULL |

GASPI_QUEUE_FULL is returned if the end of the used queue has been reached.

3.3 Boilerplate

If the queue to which the communication request has been 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 has not been issued and the procedure returns with return value GASPI_QUEUE_FULL. In this case users should either switch to another queue or wait (see gaspi_wait) and subsequently re-issue the communication request.

3.4 BoilerplateWriteList

If the queue to which the communication request has been 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 has not been issued and the procedure returns with return value GASPI_QUEUE_FULL. In this case users should either switch to another queue or wait (see gaspi_wait) and subsequently re-issue the communication request.

The user should be aware that a subsequent gaspi_notify only guarantees non-overtaking conditions for the same queue to which previous gaspi_write calls have been posted.

3.5 Functions

| 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

```fortran
module gaspi

contains

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 )

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_QUEUE_FULL: operation could not be posted due to a full queue

The listings for alltoall_write and alltoall_read should change accordingly:

Listing 1: wait_if_queue_full.h

```c
#ifndef _WAIT_IF_QUEUE_FULL_H
#define _WAIT_IF_QUEUE_FULL_H

#endif

```

---

3
# include <GASPI.h>

#define WAIT_IF_QUEUE_FULL(f, queue) \
{ \
    gaspi_return_t ret; \
    while ((ret = (f)) == GASPI_QUEUE_FULL) \
    { \
        ASSERT (gaspi_wait ((queue), GASPI_BLOCK)); \
    } \
    ASSERT (ret == GASPI_SUCCESS); \
} \
#endif

Listing 2: 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;

#pragma omp parallel
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;
    const gaspi_notification_t notify_val = 1;

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

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 3: GASPI all-to-all communication (matrix transpose) implemented with gaspi

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

    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; 
}  