Proposal to the GASPI Specification
Inclusion of gaspi_read_notify

Christian Simmendinger and Vanessa End
January 25, 2016

Contents

1 Introduction and Motivation .................................................. 1

1.0.1 gaspi_read_notify .................................................. 4

2 Needed Resources .................................................. 6

3 Additional (necessary) Changes to the Standard .................. 6

1 Introduction and Motivation

Moving towards the exascale era in high performance computing we see the necessity to include a notification driven gaspi_read_notify routine into the GASPI standard, which complements the existing 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 this proposal 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 (\(\text{queue\_num} \times \text{queue\_size\_max}\)).

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.
The two GASPI functions \texttt{gaspi\_read\_notify} and \texttt{gaspi\_waitsome} establish a logical and thread safe happens-before relation between them. Since hitherto \texttt{gaspi\_read} and \texttt{gaspi\_wait} have to be issued by the same thread, the procedure \texttt{gaspi\_read\_notify} significantly extends the general applicability of remote read operations.

A typical use of \texttt{gaspi\_read\_notify} takes the following form:

Listing 1: \texttt{gaspi\_read\_notify} Example usage

```c
// 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 resources 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 ));

1.0.1 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 preceding read operations. For this latter functionality a call to the gaspi_wait procedure is required.
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 preceding 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.

2 Needed Resources
  • none.

3 Additional (necessary) Changes to the Standard
  • 8.3.3
    For the procedures with notification, `gaspi_notify` and the extendend functions `gaspi_write_notify` and `gaspi_read_notify`, the function `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`).

  • additional user advice

    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_read_notify` has arrived as well.