GASPI Tutorial

Christian Simmendinger

Mirko Rahn

Daniel Grünewald

Goals

- Get an overview over GASPI

- Learn how to
  - Compile a GASPI program
  - Execute a GASPI program

- Get used to the GASPI programming model
  - one-sided communication
  - weak synchronization
  - asynchronous patterns / dataflow implementations

Outline

- Introduction to GASPI

- GASPI API

  – Execution model

  – Memory segments

  – One-sided communication

  – Collectives

  – Passive communication

Outline

- GASPI programming model
  - Dataflow model
  - Fault tolerance

www.gaspi.de   **www.gpi-site.com**
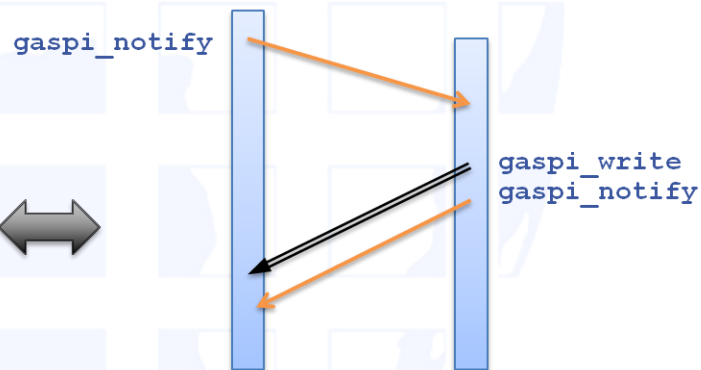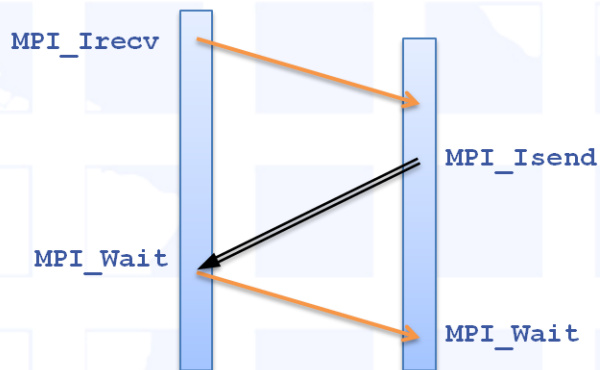
Introduction to GASPI

Motivation

- **A PGAS API for SPMD execution**
- **Take your existing MPI code**
- **Rethink your communication patterns !**
- **Reformulate towards an asynchronous data flow model !**

Bulk-synchronous Execution ⟷ Asynchronous Execution

MPI_Irecv

MPI_Isend

MPI_Wait

MPI_Wait

gaspi_notify

gaspi_write
gaspi_notify

Key Objectives of GASPI

- **Scalability**
  - From bulk–synchronous two sided communication patterns to asynchronous one-sided communication
  - remote completion

- **Flexibility and Versatility**
  - Multiple Segments,
  - Configurable hardware ressources
  - Support for multiple memory models

- **Failure Tolerance**
  - Timeouts in non-local operations
  - dynamic node sets.
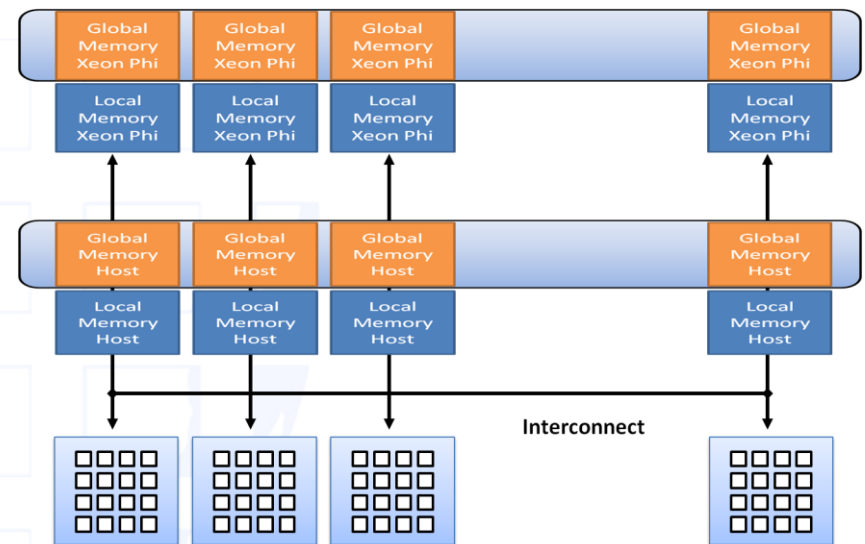
GASPI history

- **GPI**
  - originally called Fraunhofer Virtual Machine (**FVM**)
  - developed since 2005
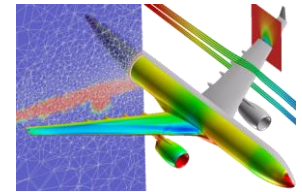 - used in many of the industry projects at CC-HPC of Fraunhofer ITWM

GPI: Winner of the „Joseph von Fraunhofer Preis 2013"

www.gpi-site.com

Scalability

Performance

- One-sided read and writes

- **remote completion in PGAS** with notifications.

- Asynchronous execution model

  - **RDMA queues** for one-sided read and write operations, including support for arbitrarily distributed data.

- Threadsafety

  - Multithreaded communication is the default rather than the exception.

- Write, Notify, Write_Notifiy

  - **relaxed synchronization** with double buffering

  - traditional (asynchronous) handshake mechanisms remain possible.

- No Buffered Communication - Zero Copy.

Scalability
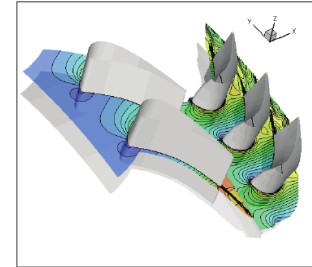
Performance

- No polling for outstanding receives/acknowledges for send
  - **no communication overhead**, true asynchronous RDMA read/write.
- Fast synchronous collectives with time-based blocking and timeouts
  - Support for asynchronous collectives in core API.
- Passive Receives two sided semantics, no Busy-Waiting
  - Allows for distributed updates, non-time critical asynchronous collectives. Passive Active Messages, so to speak ☺.
- Global Atomics for all data in segments
  - FetchAdd
  - cmpSwap.
- Extensive profiling support.
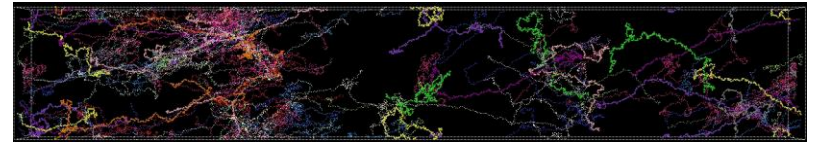
Flexibility and Versatility

- Segments
  - Support for **heterogeneous Memory Architectures** (NVRAM, GPGPU, Xeon Phi, Flash devices).
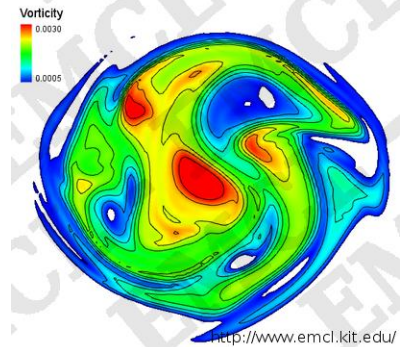 - Tight coupling of Multi-Physics Solvers
  - Runtime evaluation of applications (e.g Ensembles)
- Multiple memory models
  - Symmetric Data Parallel (OpenShmem)
  - Symmetric Stack Based Memory Management
  - Master/Slave
  - Irregular.

Vorticity
0.0030
0.0005

http://www.emcl.kit.edu/
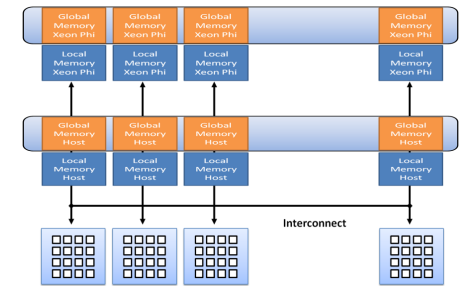
Flexibility

Interoperability and Compatibility

- Compatibility with most Programming Languages.

- Interoperability with MPI.

- Compatibility with the Memory Model of OpenShmem.

- Support for all Threading Models (OpenMP/Pthreads/..)

  - similar to MPI, GASPI is orthogonal to Threads.

- GASPI is a nice match for **tile architecture** with **DMA** engines.
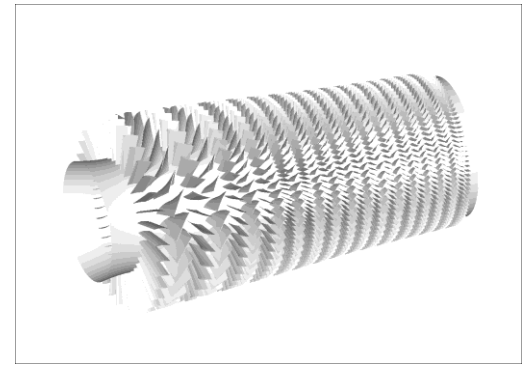
Flexibility

Flexibility

- Allows for **shrinking and growing** node set.

- User defined global reductions with **time based blocking**.

- Offset lists for RDMA read/write (write_list, write_list_notify)

- **Groups** (Communicators)

- Advanced Ressource Handling, configurable setup at startup.

- Explicit connection management.
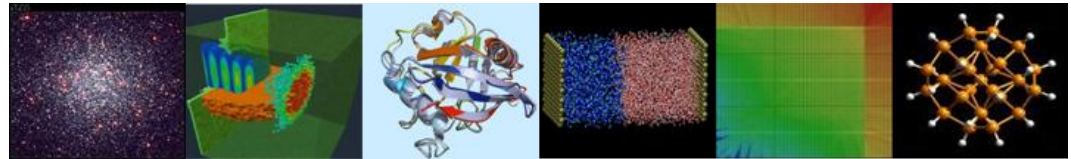
Failure Tolerance

Failure Tolerance.

- Timeouts in all non-local operations

- Timeouts for Read, Write, Wait, Segment Creation, Passive Communication.

- Dynamic growth and shrinking of node set.

- Fast Checkpoint/Restarts to NVRAM.

- State vectors for GASPI processes.

The GASPI API

- 52 communication functions
- 24 getter/setter functions
- 108 pages
  … but in reality:
  - Init/Term
  - Segments
  - Read/Write
  - Passive Communication
  - Global Atomic Operations
  - Groups and collectives

www.gaspi.de

```
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

Execution Model

GASPI Exection Model

- SPMD / MPMD execution model
- All procedures have prefix gaspi_

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

- All procedures have a return value
- Timeout mechanism for potentially blocking procedures

GASPI Return Values

- Procedure return values:
  - GASPI_SUCCESS
    - designated operation successfully completed
  - GASPI_TIMEOUT
    - designated operation could not be finished in the given period of time
    - not necessarily an error
    - the procedure has to be invoked subsequently in order to fully complete the designated operation
  - GASPI_ERROR
    - designated operation failed -> check error vector
- Advice: Always check return value !

success_or_die.h

```c
#ifndef SUCCESS_OR_DIE_H
#define SUCCESS_OR_DIE_H

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

#define SUCCESS_OR_DIE(f...)              \
do                                        \
{                                         \
  const gaspi_return_t r = f;        \
                                          \
  if (r != GASPI_SUCCESS)            \
  {                                       \
    gaspi_printf ("Error: '%s' [%s:%i]: %i\n", #f, __FILE__, __LINE__, r);\
                                          \
      exit (EXIT_FAILURE);         \
   }                                      \
} while (0)

#endif
```

Timeout Mechanism

- Mechanism for potentially blocking procedures
  - procedure is guaranteed to return
- Timeout: gaspi_timeout_t
  - GASPI_TEST (0)
    - procedure completes local operations
    - Procedure does not wait for data from other processes
  - GASPI_BLOCK (-1)
    - wait indefinitely (blocking)
  - Value > 0
    - Maximum time in msec the procedure is going to wait for data from other ranks to make progress
    - != hard execution time

GASPI Process Management

- Initialize / Finalize
  - gaspi_proc_init
  - gaspi_proc_term
- Process identification
  - gaspi_proc_rank
  - gaspi_proc_num
- Process configuration
  - gaspi_config_get
  - gaspi_config_set

GASPI Initialization

- ## gaspi_proc_init

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

- – initialization of resources
  - set up of communication infrastructure if requested
  - set up of default group GASPI_GROUP_ALL
  - rank assignment
    - – position in machinefile ⇔ rank ID
- – no default segment creation

GASPI Finalization

- ## gaspi_proc_term

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

  – clean up
    - wait for outstanding communication to be finished
    - release resources
  – no collective operation !

GASPI Process Identification

- gaspi_proc_rank

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

- gaspi_proc_num

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

GASPI Process Configuration

- gaspi_config_get

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

- gaspi_config_set

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

- Retrieveing and setting the configuration structure has to be done before gaspi_proc_init

GASPI Process Configuration

- Configuring
  - resources
    - sizes
    - max
  - network

```c
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;
```

GASPI „hello world"

```c
#include "success_or_die.h"
#include <GASPI.h>
#include <stdlib.h>

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

  gaspi_rank_t rank;
  gaspi_rank_t num;
  SUCCESS_OR_DIE( gaspi_proc_rank(&rank) );
  SUCCESS_OR_DIE( gaspi_proc_num(&num) );

  gaspi_printf("Hello world from rank %d of %d\n",rank, num);

  SUCCESS_OR_DIE( gaspi_proc_term(GASPI_BLOCK) );
  return EXIT_SUCCESS;
}
```
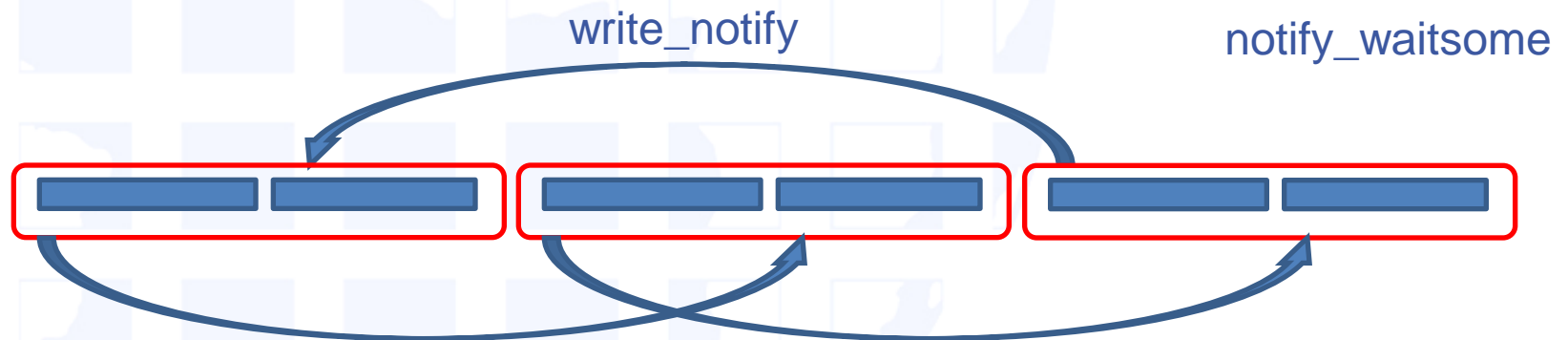
Communication example

- init local buffer
- write to remote buffer
- wait for data availability
- print

onesided.c (I)

```c
// includes

int main(int argc, char *argv[])
{
  static const int VLEN = 1 << 2;
  SUCCESS_OR_DIE( gaspi_proc_init(GASPI_BLOCK) );
  gaspi_rank_t iProc, nProc;
  SUCCESS_OR_DIE( gaspi_proc_rank(&iProc));
  SUCCESS_OR_DIE( gaspi_proc_num(&nProc));
  gaspi_segment_id_t const segment_id = 0;
  gaspi_size_t       const segment_size = 2 * VLEN * sizeof (double);

  SUCCESS_OR_DIE ( gaspi_segment_create ( segment_id, segment_size
                                    , GASPI_GROUP_ALL, GASPI_BLOCK
                                    , GASPI_MEM_UNINITIALIZED ) );
  gaspi_pointer_t array;
  SUCCESS_OR_DIE ( gaspi_segment_ptr (segment_id, &array) );
  double * src_array = (double *)(array);
  double * rcv_array = src_array + VLEN;

  for (int j = 0; j < VLEN; ++j) {
    src_array[j]= (double)( iProc * VLEN + j ); }
```

```
  gaspi_notification_id_t data_available = 0;
  gaspi_queue_id_t queue_id = 0;
  gaspi_offset_t loc_off = 0;
  gaspi_offset_t rem_off = VLEN * sizeof (double);

  wait_for_queue_entries_for_write_notify ( &queue_id );
  SUCCESS_OR_DIE ( gaspi_write_notify ( segment_id, loc_off
                                     , RIGHT (iProc, nProc)
                                     , segment_id, rem_off
                                     , VLEN * sizeof (double)
                                     , data_available, 1 + iProc, queue_id
                                     , GASPI_BLOCK )  );
  wait_or_die (segment_id, data_available, 1 + LEFT (iProc, nProc) );

  for (int j = 0; j < VLEN; ++j)
  { gaspi_printf("rank %d rcv elem %d: %f \n", iProc,j,rcv_array[j] );     }

  wait_for_flush_queues();
  SUCCESS_OR_DIE( gaspi_proc_term(GASPI_BLOCK) );
  return EXIT_SUCCESS;
}
```

Memory Segments

Segments

- software abstraction of hardware memory hierarchy
  - NUMA
  - GPU
  - Xeon Phi
- one partition of the PGAS
- contiguous block of virtual memory
  - no pre-defined memory model
  - memory management up to the application
- locally / remotely accessible
  - local access by ordinary memory operations
 - remote access by GASPI communication routines

GASPI Segments

- GASPI provides only a few relatively large segments
  - segment allocation is expensive
  - the total number of supported segments is limited by hardware constraints
- GASPI segments have an allocation policy
  - GASPI_MEM_UNINITIALIZED
    - memory is not initialized
  - GASPI_MEM_INITIALIZED
    - memory is initialized (zeroed)

Segment Functions

- Segment creation
  - gaspi_segment_alloc
  - gaspi_segment_register
  - gaspi_segment_create
- Segment deletion
  - gaspi_segment_delete
- Segment utilities
  - gaspi_segment_num
  - gaspi_segment_ptr
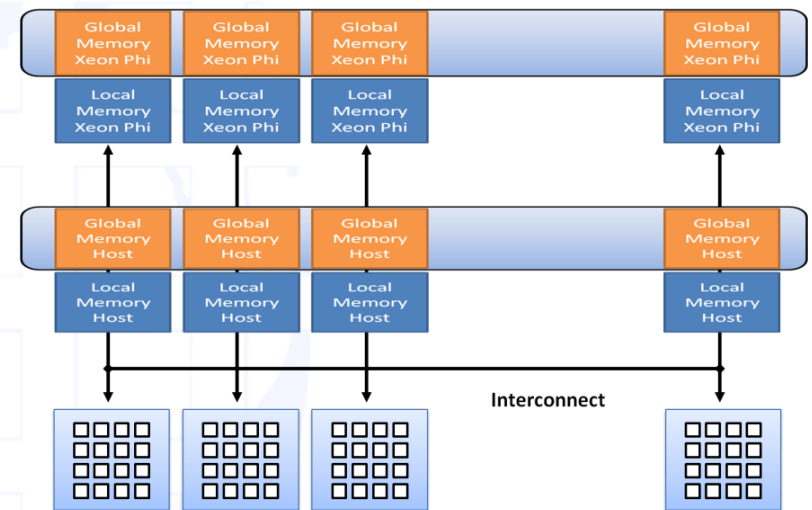
GASPI Segment Allocation

- gaspi_segment_alloc

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

  - allocate and pin for RDMA

  - Locally accessible

- gaspi_segment register

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

  - segment accessible by rank

GASPI Segment Creation

- gaspi_segment_create

```
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 )
```

  – Collective short cut to
    - gaspi_segment_alloc
    - gaspi_segment_register
  – After successful completion, the segment is locally and remotely accessible by all ranks in the group

GASPI Segment Deletion

- gaspi_segment_delete

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

  – free segment memory

GASPI Segment Utils

- gaspi_segment_num

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

- gaspi_segment_list

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

- gaspi_segment_ptr

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

One-sided Communication

GASPI One-sided Communication

- gaspi_write

```
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 )
```

  – Post a put request into a given queue for transfering data from a local segment into a remote segment

GASPI One-sided Communication

- gaspi_read

```
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 )
```

  – Post a get request into a given queue for transfering data
    from a remote segment into a local segment

GASPI One-sided Communication

- gaspi_wait

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

- wait on local completion of all requests in a given queue
- After successfull completion, all involved local buffers are valid

Queues (I)

- Different queues available to handle the communication requests
- Requests to be submitted to one of the supported queues
- Advantages
  - more scalability
  - channels for different types of requests
  - similar types of requests are queued and synchronized together but independently from other ones
  - separation of concerns

Queues (II)

- Fairness of transfers posted to different queues is guaranteed
  - No queue should see ist communication requests delayed indefinitely
- A queue is identified by its ID
- Synchronization of calls by the queue
- Queue order does not imply message order on the network / remote memory
- A subsequent notify call is guaranteed to be non-overtaking for all previous posts to the same queue and rank

Weak Synchronization

- One sided-communication:
  - Entire communication managed by the local process only
  - Remote process is not involved
  - Advantage: no inherent synchronization between the local and the remote process in every communication request
- Still: At some point the remote process needs knowledge about data availability
  - Managed by weak synchronization primitives

Weak Synchronization

- Several notifications for a given segment
  - Identified by notification ID
  - Logical association of memory location and notification

GASPI Weak Synchronization

- gaspi_notify

```
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 )
```

  – posts a notification with a given value to a given queue

  – remote visibility guarantees remote data visibility of all previously posted writes in the same queue, the same segment and the same process rank

GASPI Weak Synchronization

- gaspi_notify_waitsome

```
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 )
```

– monitors a contiguous subset of notification id's for a given segment

– returns successfull if at least one of the monitored id's is remotely updated to a value unequal zero

GASPI Weak Synchronization

- gaspi_notify_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)
```

- – Atomically resets a given notification id and yields the old value

Extended One-sided Calls

- gaspi_write_notify
  - gaspi_write + subsequent gaspi_notify
- gaspi_write_list
  - several subsequent gaspi_writes to the same rank
- gaspi_write_list_notify
  - gaspi_write_list + subsequent gaspi_notify
- gaspi_read_list
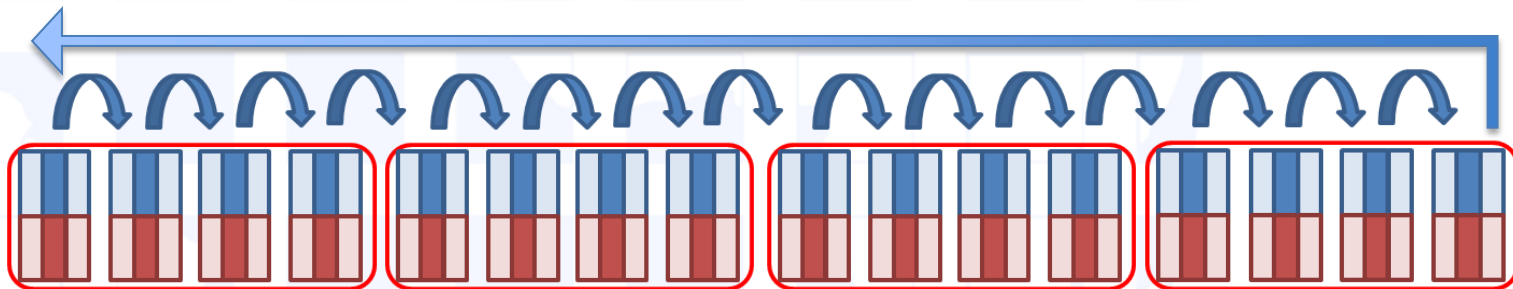 - several subsequent gaspi_reads

Dataflow model

Hands On Session
The MPI/GASPI Ring Exchange

The MPI Ring Exchange

MPI – MPI_Issend/MPI_Recv

- NITER iterations of Ring Exchange with „nProc" cores
- Shift upper half of vector to the right

Example: 4 Sockets/16 cores – each core holds a vector of length 2*VLEN

The MPI Ring Exchange

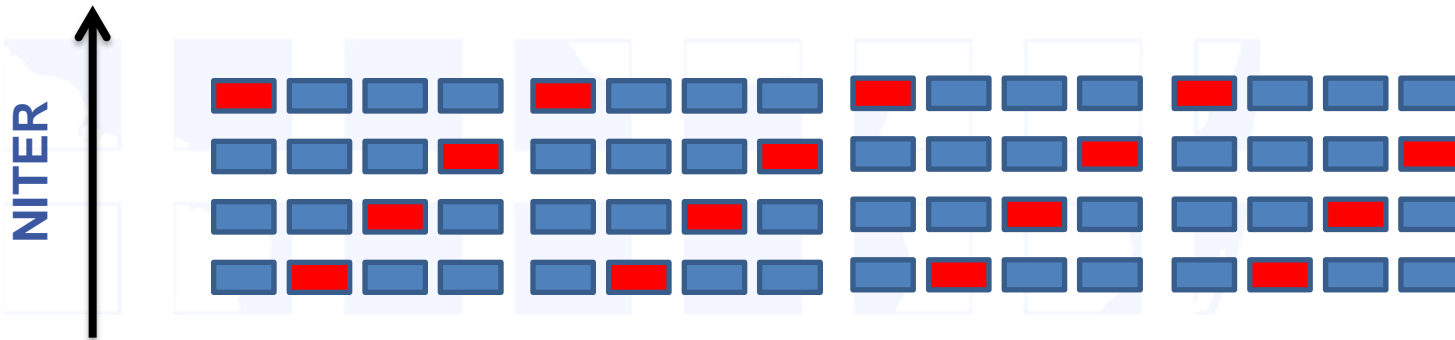## MPI – round_robin_double_buffer.c

```c
for (int i = 0; i < nProc; ++i) {
    MPI_Request request;
    int left_halo   = 0;
    int slice_id    = 1;
    // issue send
    MPI_Isend ( &array_ELEM_right (buffer_id, slice_id, 0), VLEN,
        MPI_DOUBLE, right, i, MPI_COMM_WORLD, &request);
    // post recv
    MPI_Recv ( &array_ELEM_right (buffer_id, left_halo, 0), VLEN,
        MPI_DOUBLE, left, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // compute data, read from id "buffer_id", write to id "1 - buffer_id"
    data_compute (NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
    // wait for Issend
    MPI_Wait (&request, MPI_STATUS_IGNORE);
    // alternate the buffers
    buffer_id = 1 - buffer_id;
}
```

The MPI Ring Exchange

MPI – MPI_Issend/MPI_Recv

- Synchronous Lockstep Execution

The MPI Ring Exchange

MPI – left_right_double_buffer.c

```c
for (int i = 0; i < nProc; ++i) {
    MPI_Request send_req[2], recv_req[2];
    const int left_halo   = 0; slice_id   = 1; right_halo  = 2;
    MPI_Irecv ( &array_ELEM_right (buffer_id, left_halo, 0), VLEN,
        MPI_DOUBLE, left, i, MPI_COMM_WORLD, &send_req[0]);
    MPI_Irecv ( &array_ELEM_left (buffer_id, right_halo, 0), VLEN,
        MPI_DOUBLE, right, i, MPI_COMM_WORLD, &send_req[1]);
    MPI_Isend ( &array_ELEM_right (buffer_id, slice_id, 0), VLEN,
        MPI_DOUBLE, right, i, MPI_COMM_WORLD, &recv_req[0]);
    MPI_Isend ( &array_ELEM_left (buffer_id, slice_id, 0), VLEN,
        MPI_DOUBLE, left, i, MPI_COMM_WORLD, &recv_req[1]);
    MPI_Waitall (2, recv_req, MPI_STATUSES_IGNORE);
    data_compute (NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
    MPI_Waitall (2, send_req, MPI_STATUSES_IGNORE);
    buffer_id = 1 - buffer_id;
}
```

The MPI Ring Exchange

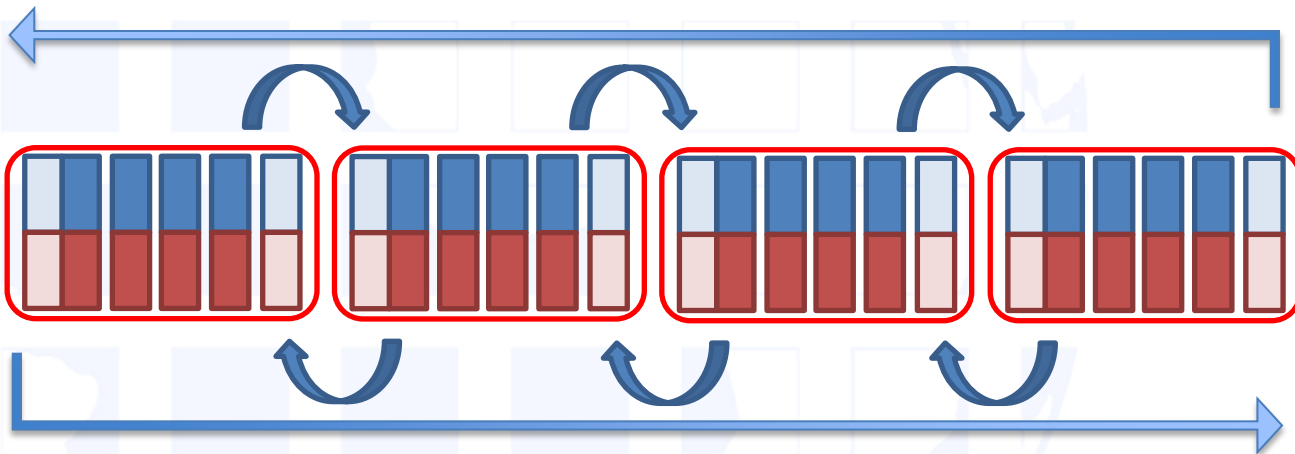## MPI – left_right_double_buffer_req_free.c

```c
for (int i = 0; i < nProc; ++i) {
    MPI_Request send_req[2], recv_req[2];
    const int left_halo  = 0; slice_id   = 1; right_halo  = 2;
    MPI_Irecv ( &array_ELEM_right (buffer_id, left_halo, 0), VLEN,
      MPI_DOUBLE, left, i, MPI_COMM_WORLD, &send_req[0]);
    MPI_Irecv ( &array_ELEM_left (buffer_id, right_halo, 0), VLEN,
      MPI_DOUBLE, right, i, MPI_COMM_WORLD, &send_req[1]);
    MPI_Isend ( &array_ELEM_right (buffer_id, slice_id, 0), VLEN,
      MPI_DOUBLE, right, i, MPI_COMM_WORLD, &recv_req[0]);
    MPI_Isend ( &array_ELEM_left (buffer_id, slice_id, 0), VLEN,
      MPI_DOUBLE, left, i, MPI_COMM_WORLD, &recv_req[1]);
    MPI_Request_free(&send_req[0]);
    MPI_Request_free(&send_req[1]);
    MPI_Waitall (2, recv_req, MPI_STATUSES_IGNORE);
    data_compute (NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
    buffer_id = 1 - buffer_id;
  }
```

The MPI Ring Exchange

MPI – **HYBRID MPI/OpenMP**

- Shift upper half of the vector to the right
- Shift lower half of the vector to the left

Example: 4 Sockets/16 cores – each core holds a vector of length 2*VLEN

The MPI Ring Exchange

- MPI – left_right_double_buffer_funneled.c

```
for ( int i = 0; i < nProc * NTHREADS; ++i ) {
    const int left_halo = 0, slice_id = tid + 1, right_halo  = NTHREADS+1;
    if (tid == 0) {
      MPI_Request send_req[2], recv_req[2];
      MPI_Irecv ( &array_ELEM_right (buffer_id, left_halo, 0), VLEN,
          MPI_DOUBLE, left, i, MPI_COMM_WORLD, &recv_req[0]);
      MPI_Irecv ( &array_ELEM_left (buffer_id, right_halo, 0), VLEN,
          MPI_DOUBLE, right, i, MPI_COMM_WORLD, &recv_req[1]);
      MPI_Isend ( &array_ELEM_right (buffer_id, slice_id, 0), VLEN,
          MPI_DOUBLE, right, i, MPI_COMM_WORLD, &send_req[0]);
      MPI_Isend ( &array_ELEM_left (buffer_id, slice_id, 0), VLEN,
          MPI_DOUBLE, left, i, MPI_COMM_WORLD, &send_req[1]);
      MPI_Request_free(&send_req[0]);
      MPI_Request_free(&send_req[1]);
      MPI_Waitall (2, recv_req, MPI_STATUSES_IGNORE);
    }
#pragma omp barrier
    data_compute (NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
#pragma omp barrier
    buffer_id = 1 - buffer_id; }
```

The MPI Ring Exchange

MPI – left_right_double_buffer_funneled.c

- Fork-join model
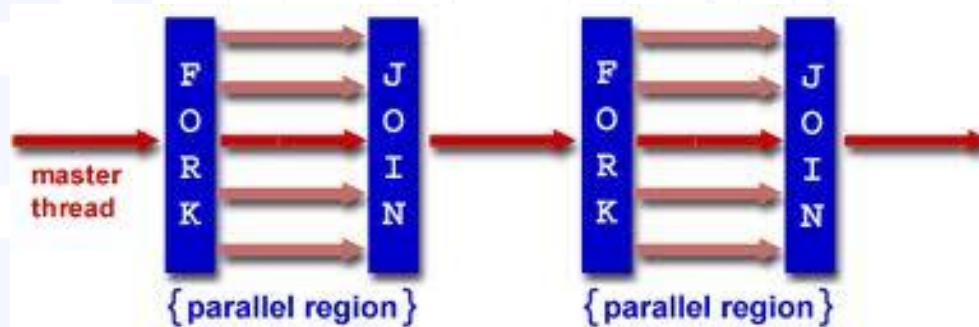
The MPI Ring Exchange

- MPI – left_right_double_buffer_multiple.c

```
    if (tid == 0) {
      MPI_Request request;
      MPI_Isend ( &array_ELEM_left (buffer_id, slice_id, 0), VLEN,
        MPI_DOUBLE, left, i, MPI_COMM_WORLD, &request);
      MPI_Request_free(&request);
      MPI_Recv ( &array_ELEM_right (buffer_id, left_halo, 0), VLEN,
        MPI_DOUBLE, left, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      data_compute (NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
} else if (tid < NTHREADS - 1){
      data_compute (NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
    } else {
      MPI_Request request;
      MPI_Isend ( &array_ELEM_right (buffer_id, slice_id, 0), VLEN,
        MPI_DOUBLE, right, i, MPI_COMM_WORLD, &request);
      MPI_Request_free(&request);
      MPI_Recv ( &array_ELEM_left (buffer_id, right_halo, 0), VLEN,
        MPI_DOUBLE, right, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      data_compute (NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
    }
#pragma omp barrier
    buffer_id = 1 - buffer_id;
```
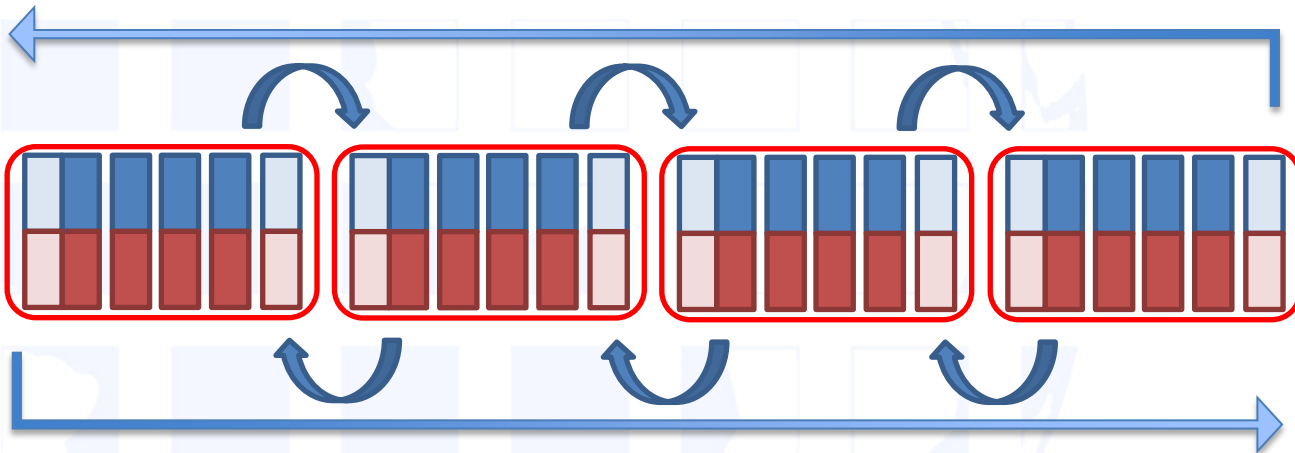
The GASPI Ring Exchange

GASPI – **HYBRID GASPI/OpenMP**

- Shift upper half of the vector to the right
- Shift lower half of the vector to the left

Example: 4 Sockets/16 cores – each core holds a vector of length 2*VLEN

The GASPI Ring Exchange
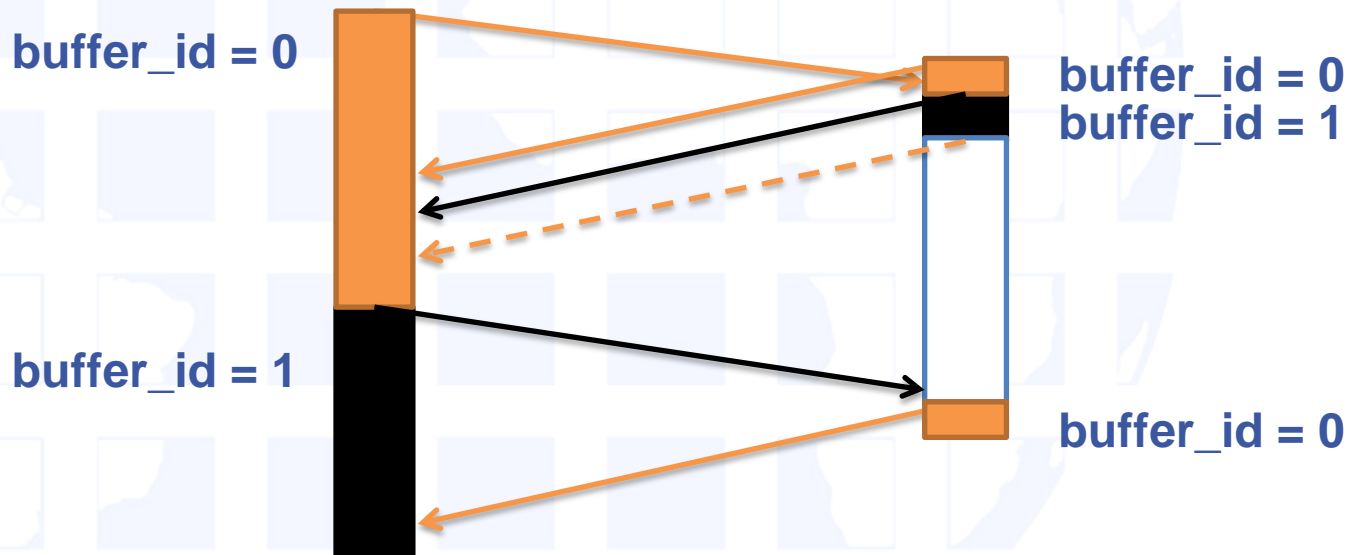
- GASPI – left_right_double_buffer_funneled.c

```
if (tid == 0) {
 wait_for_queue_max_half (&queue_id);
 SUCCESS_OR_DIE ( gaspi_write_notify
    ( segment_id,array_OFFSET_left(buffer_id, slice_id, 0), left,
    segment_id,array_OFFSET_left(buffer_id,right_halo,0),VLEN* sizeof(double),
    right_data_available[buffer_id], 1 + i, queue_id, GASPI_BLOCK));
 wait_for_queue_max_half (&queue_id);
 SUCCESS_OR_DIE ( gaspi_write_notify
    ( segment_id, array_OFFSET_right (buffer_id, slice_id, 0), right,
    segment_id,array_OFFSET_right(buffer_id,left_halo,0),VLEN*sizeof (double),
    left_data_available[buffer_id], 1 + i, queue_id, GASPI_BLOCK));
 wait_or_die (segment_id, right_data_available[buffer_id], 1 + i);
 wait_or_die (segment_id, left_data_available[buffer_id], 1 + i);
 }
#pragma omp barrier
data_compute ( NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
#pragma omp barrier
buffer_id = 1 - buffer_id;
```

The GASPI Ring Exchange

GASPI – left_right_double_buffer_funneled.c

- Bi-directional halo exchange –

    implicit synchronization

The GASPI Ring Exchange

- GASPI – left_right_double_buffer_multiple.c

```c
if (tid == 0) {
  wait_for_queue_max_half (&queue_id);
  SUCCESS_OR_DIE ( gaspi_write_notify
    (segment_id, array_OFFSET_left (buffer_id, slice_id, 0), left,
     segment_id,array_OFFSET_left(buffer_id, right_halo,0),VLEN*sizeof(double),
     right_data_available[buffer_id], 1 + i, queue_id, GASPI_BLOCK));
  wait_or_die (segment_id, left_data_available[buffer_id], 1 + i);
  data_compute ( NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
} else if (tid < NTHREADS - 1) {
  data_compute ( NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
} else {
  wait_for_queue_max_half (&queue_id);
  SUCCESS_OR_DIE ( gaspi_write_notify
    ( segment_id, array_OFFSET_right (buffer_id, slice_id, 0), right,
     segment_id,array_OFFSET_right(buffer_id,left_halo, 0),VLEN*sizeof(double),
     left_data_available[buffer_id], 1 + i, queue_id, GASPI_BLOCK));
  wait_or_die (segment_id, right_data_available[buffer_id], 1 + i);
  data_compute ( NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
}
#pragma omp barrier
buffer_id = 1 - buffer_id;
```

The GASPI Ring Exchange

- GASPI – left_right_double_buffer_multiple.c

- One message instead of three (MPI Rendezvouz)
- No waiting for late MPI_Recv
- No waiting for acknowledge for MPI_Issend
- Overlap of communication with computation

The GASPI Ring Exchange

- GASPI – Dataflow - left_right_dataflow_halo.c

```
#pragma omp parallel default (none) firstprivate (buffer_id, queue_id)  \
  shared (array, data_available, ssl, stderr)
  {
    slice* sl;
    while (sl = get_slice_and_lock (ssl, NTHREADS, num))
    {
      handle_slice(sl, array, data_available, segment_id, queue_id,
        NWAY, NTHREADS, num);
      omp_unset_lock (&sl->lock);
    }
  }
```

```
typedef struct slice_t
{
  omp_lock_t lock;
  volatile int stage;
  int index;
  enum halo_types halo_type;
  struct slice_t *left;
  struct slice_t *next;
} slice;
```

The GASPI Ring Exchange

- GASPI – Dataflow - slice.c

```c
void handle_slice ( …)
 if (sl->halo_type == LEFT){
    if (sl->stage > sl->next->stage) {return;}
    if (! test_or_die (segment_id, left_data_available[old_buffer_id], 1))
    { return; }
  } else if (sl->halo_type == RIGHT) {
    if (sl->stage > sl->left->stage) { return; }
    if (! test_or_die (segment_id, right_data_available[old_buffer_id], 1))
    { return; }
  } else if (sl->halo_type == NONE) {
    if (sl->stage > sl->left->stage || sl->stage > sl->next->stage) {return;}
  }
data_compute (NTHREADS, array, new_buffer_id, old_buffer_id, sl->index);
  if (sl->halo_type == LEFT) {
     SUCCESS_OR_DIE ( gaspi_write_notify …)
  } else if (sl->halo_type == RIGHT)
     SUCCESS_OR_DIE ( gaspi_write_notify …)
  }
 ++sl->stage;
}
```

The GASPI Ring Exchange

GASPI – Dataflow

- Locally and globally asynchronous dataflow.
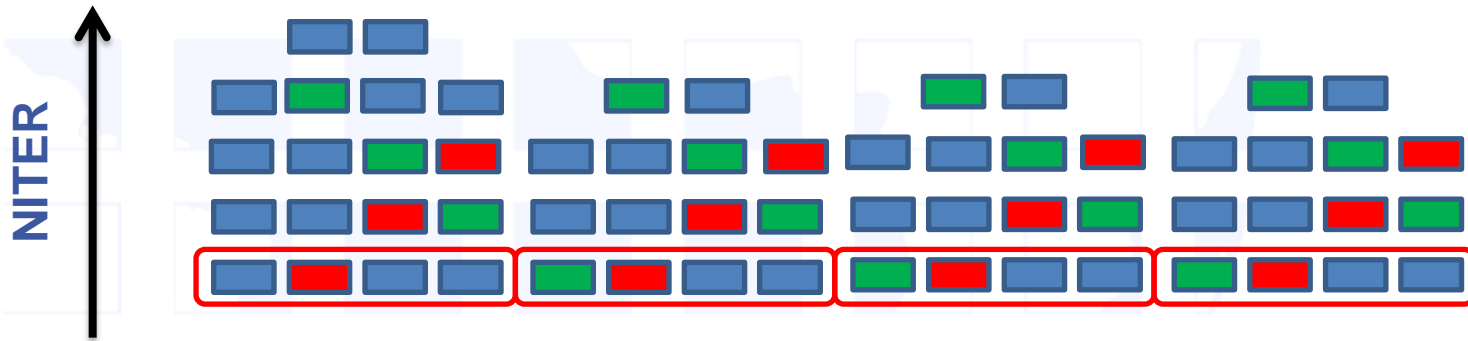
Collectives

Collective Operations (I)

- Collectivity with respect to a definable subset of ranks (groups)
  - Each GASPI process can participate in more than one group
  - Defining a group is a three step procedure
    - gaspi_group_create
    - gaspi_group_add
    - gaspi_group_commit

  - GASPI_GROUP_ALL is a predefined group containing all processes

Collective Operations (II)

- All gaspi processes forming a given group have to invoke the operation

- In case of a timeout (GASPI_TIMEOUT), the operation is continued in the next call of the procedure

- A collective operation may involve several procedure calls until completion

- Completion is indicated by return value GASPI_SUCCESS

Collective Operations (III)

- Collective operations are exclusive per group
  - Only one collective operation of a given type on a given group at a given time
  - Otherwise: undefined behaviour

- Example
  - Two allreduce operations for one group can not run at the same time
  - An allreduce operation and a barrier are allowed to run at the same time

Collective Functions

- Built in:
  - gaspi_barrier
  - gaspi_allreduce
    - GASPI_OP_MIN, GASPI_OP_MAX, GASPI_OP_SUM
    - GASPI_TYPE_INT, GASPI_TYPE_UINT, GASPI_TYPE_LONG, GASPI_TYPE_ULONG, GASPI_TYPE_FLOAT, GASPI_TYPE_DOUBLE
- User defined
  - gaspi_allreduce user

GASPI Collective Function

- gaspi_barrier

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

- gaspi_allreduce

```
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 )
```

Passive communication

Passive Communication Functions (I)

- 2 sided semantics send/recv
  - gaspi_passive_send

```
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 )
```

  - time based blocking

Passive Communication Functions (II)

– Gaspi_passive receive

```
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 )
```

- Time based blocking
- Sends calling thread to sleep
- Wakes up calling thread in case of incoming message or given timeout has been reached

Passive Communication Functions (III)

- Higher latency than one-sided comm.
  - Use cases:
    - Parameter exchange
    - management tasks

Fault Tolerance

Features

- Implementation of fault tolerance is up to the application
- But: well defined and requestable state guaranteed at any time by
  - Timeout mechanism
    - Potentially blocking routines equipped with timeout
  - Error vector
    - contains health state of communication partners
  - Dynamic node set
    - substitution of failed processes

Questions?

Thank you for your attention

www.gaspi.de   www.gpi-site.com