Global
Address Space
Programming Interface
GASPI

GASPI Tutorial

Christian Simmendinger
Mirko Rahn
Daniel Grünewald

Sponsored by the European Commission through

EXA2CT EPiGRAM INTERTWinE

Global
Address Space
Programming Interface
GASPI

Schedule

- 9:30h – 11:00h Introduction to GASPI
- 11:00h-11:30h break
- 11:30h-13:00h Segments
- 13:00h-14:00h lunch
- 14:00h-15:30h Single sided communication
- 15:30h-16:00h Break
- 16:00h-17:30h GASPI programming model
- 17:30h end

Round of Introductions

- Who are you?
- What are you doing?
- How did you get in contact with GASPI?
- What is your interest in / expectation to GASPI?

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

Global
Address Space
Programming Interface
GASPI

Outline

- GASPI programming model
  - Dataflow model
  - Fault tolerance

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

Global
Address Space
Programming Interface
GASPI

Installation

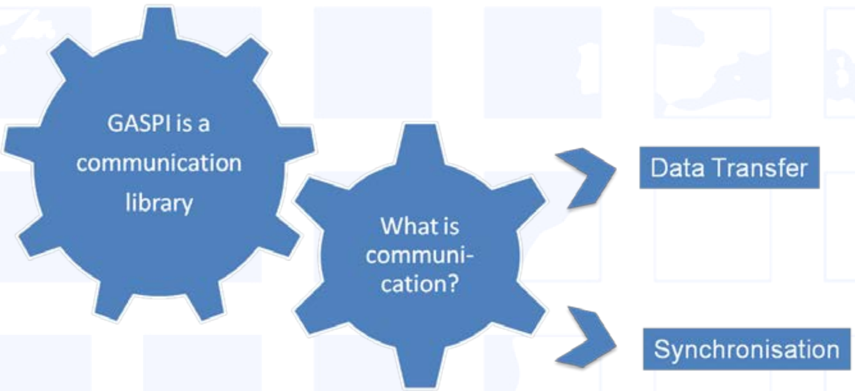
- Tutorial code and documentation:
  git clone https://github.com/GASPI-Forum/GASPI-Standard.git
- GPI-2 - GASPI Implementation:
  git clone https://github.com/cc-hpc-itwm/GPI-2.git
  - install.sh –p $HOME/GPI-2.foo
    native GASPI version, start application with gaspi_run
  - install.sh –p $HOME/GPI-2.bar –with-mpi=MPI_ROOT
    mpi interoperable version, start application with mpirun

Global
Address Space
Programming Interface
GASPI

Introduction to GASPI

Global
Address Space
Programming Interface
GASPI

GASPI at a Glance



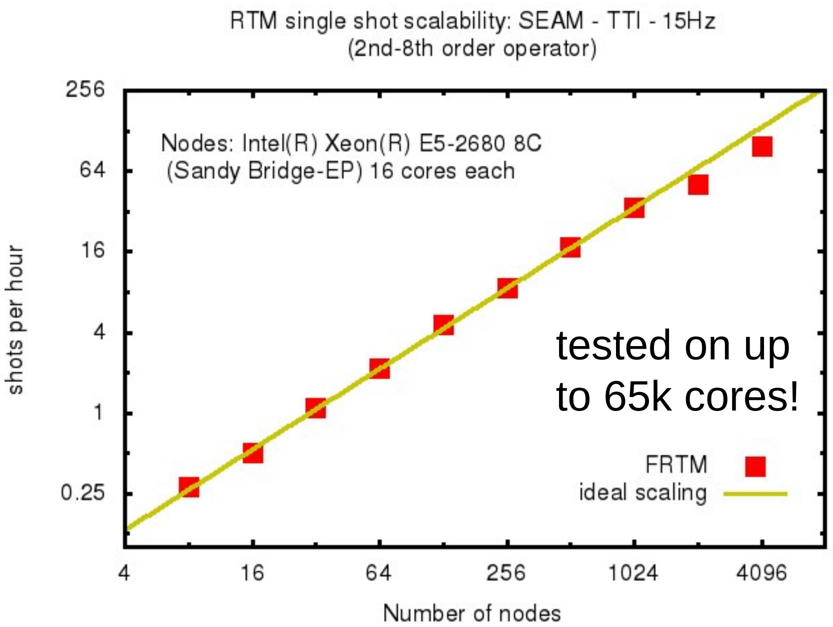
Nuts and Bolts for Communication Engines

Global
Address Space
Programming Interface
GASPI

GASPI at a Glance

Features:

- Global partitioned address space
- Asynchronous, one-sided communication
- Threadsave, every thread can communicate
- Supports fault tolerance
- Open Source
- Standardized API (GASPI)

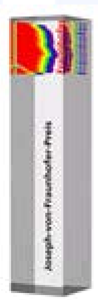


Infiniband, Cray, Ethernet, GPUs, Intel Xeon Phi, Open Source (GPL) , standardized API

Global
Address Space
Programming Interface
GASPI

GASPI History

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



Winner of the „Joseph von Fraunhofer Preis 2013"

Finalist of the „European Innovation Radar 2016".

www.gpi-site.com

Global
Address Space
Programming Interface
GASPI

GASPI Standardization Forum

Founding Members

KTH VETENSKAP OCH KONST

lrz

T · · Systems · · ·

Fraunhofer ITWM

DLR

ZIH Zentrum für Informationsdienste und Hochleistungsrechnen

ECMWF

Fraunhofer SCAI

Regionales RechenZentrum Erlangen
Der IT-Dienstleister der FAU

Global
Address Space
Programming Interface
GASPI

GASPI in
European Exascale Projects

EXascale Algorithms and Advanced Computational Techniques

Programming-model design and implementation for the Exascale

EPiGRAM

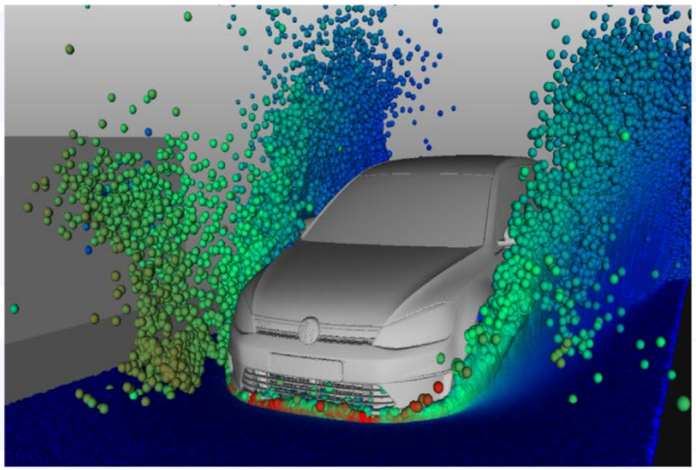
Exascale ProGRAmming Models

Inria
epcc
MANCHESTER 1824
The University of Manchester
KTH
Barcelona Supercomputing Center
Centro Nacional de Supercomputación
DLR
Fraunhofer ITWM
T-Systems

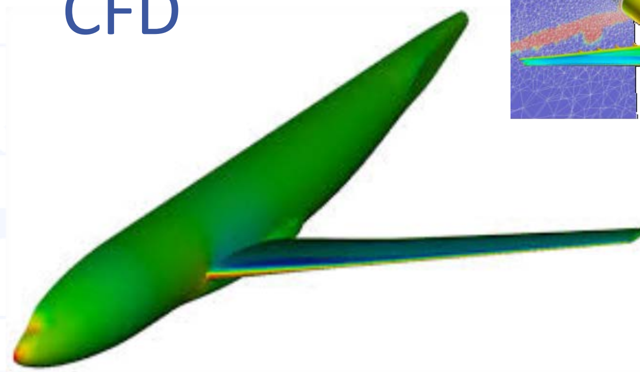
Global
Address Space
Programming Interface
GASPI

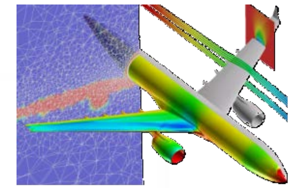
Some GASPI Applications

Visualization

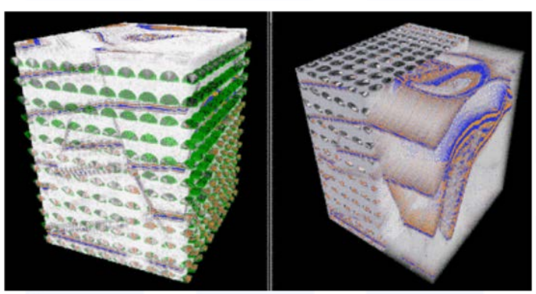


CFD

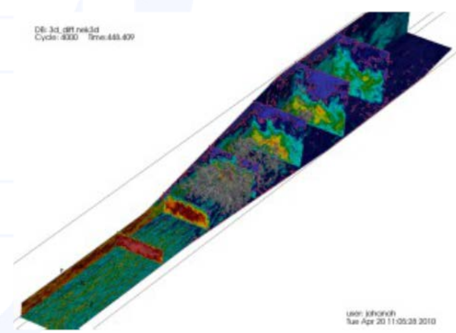




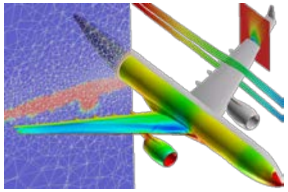
Seismic Imaging & Algorithms



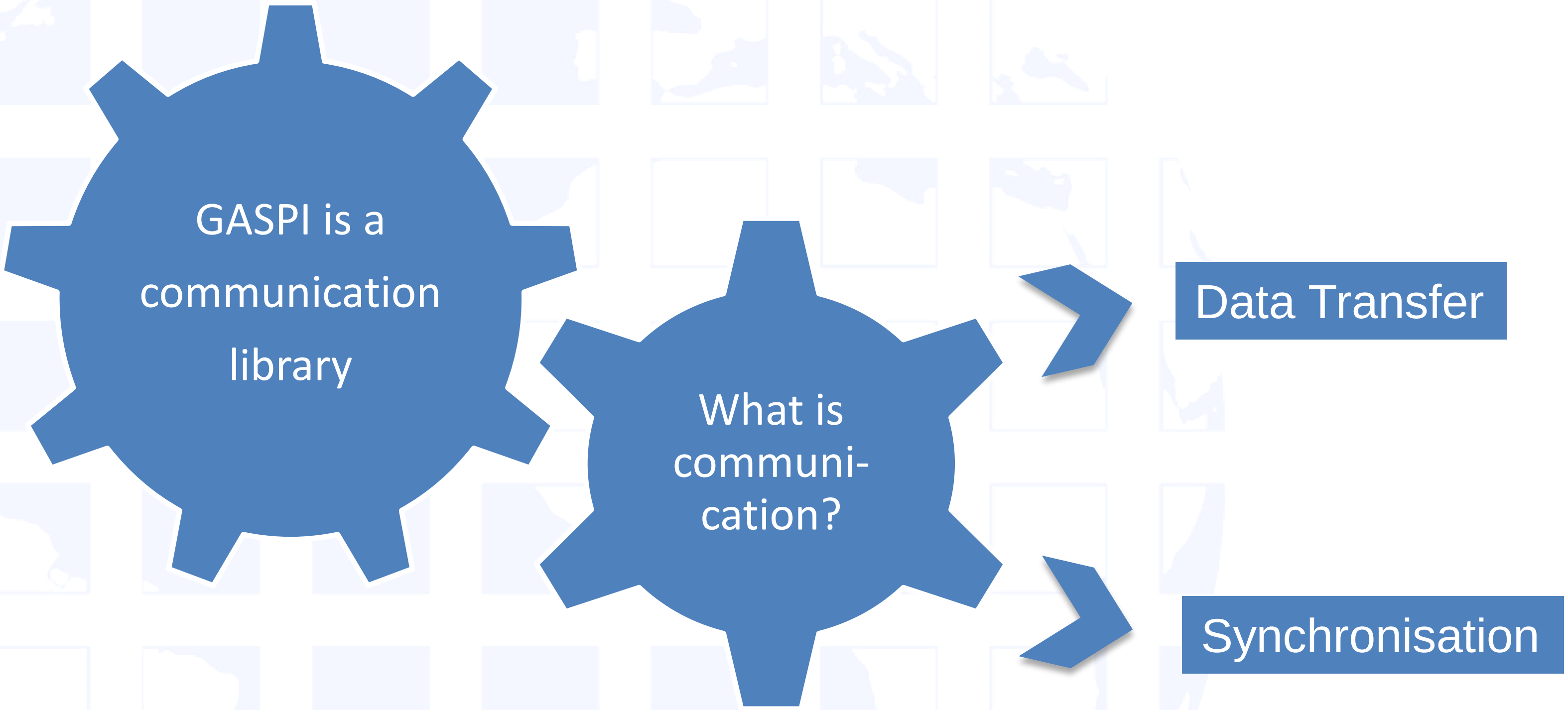
Machine Learning
Big Data
Iterative Solvers



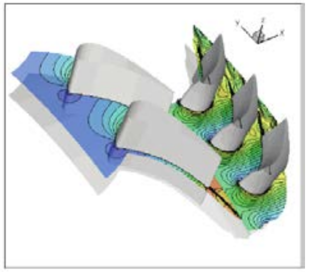
Global
Address Space
Programming Interface
GASPI



Concepts: Communication



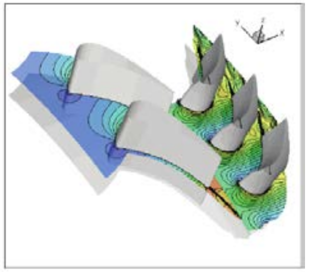
Global
Address Space
Programming Interface
GASPI



Concepts: One-Sided Communication

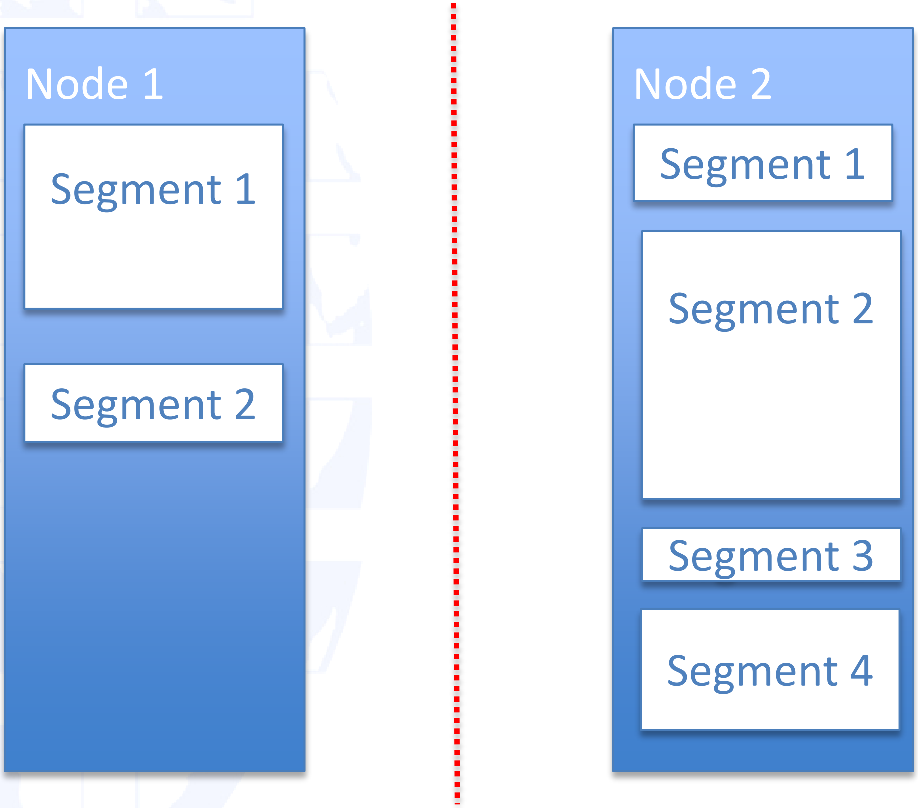
- One-sided operations between parallel processes include remote reads and writes
- Data can be accessed without participation of the remote site
- The initiator specifies all parameters
  - Source location
  - Target location
  - Message size

Global
Address Space
Programming Interface
GASPI

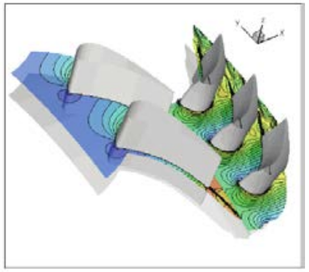


Concepts: Segments

- Data can be accessed without participation of the remote site.
- Remote sides have to know about designated communi-cation area(s) before hand
- Designated communication areas in GASPI are called segments



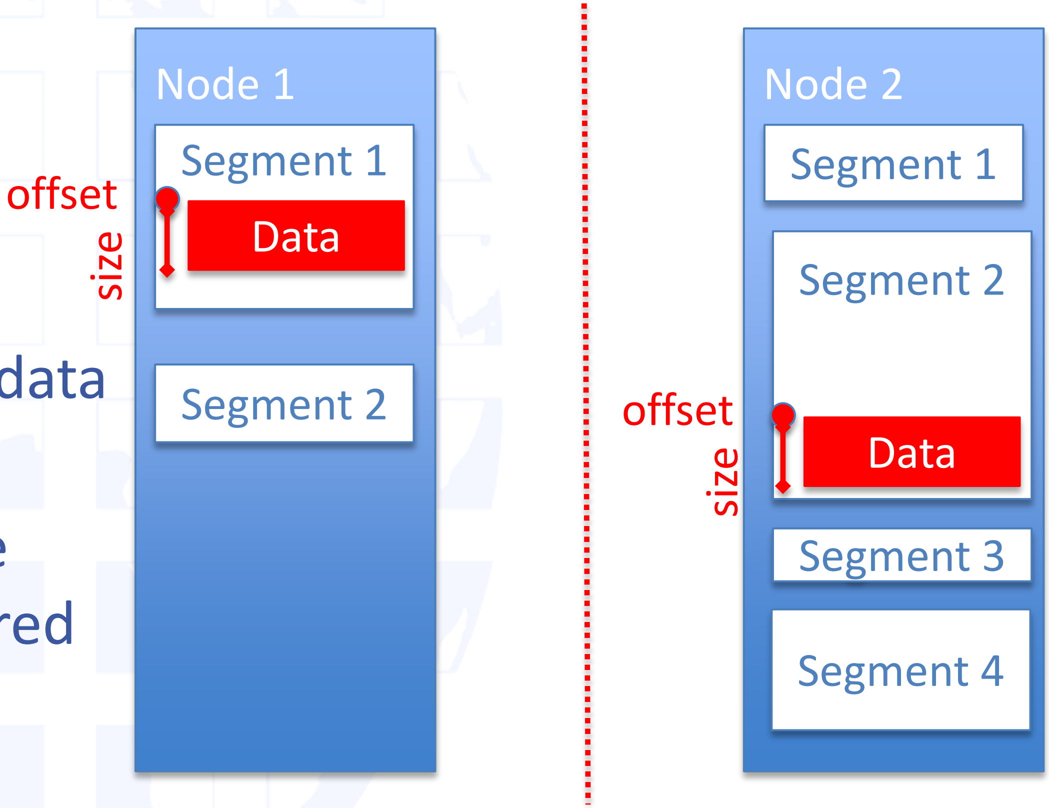
Global
Address Space
Programming Interface
GASPI



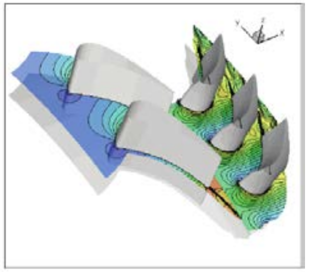
Concepts: Segments

Application has to manage data transfer completely:

- Specify which part of the segment will be transferred (offset and size)

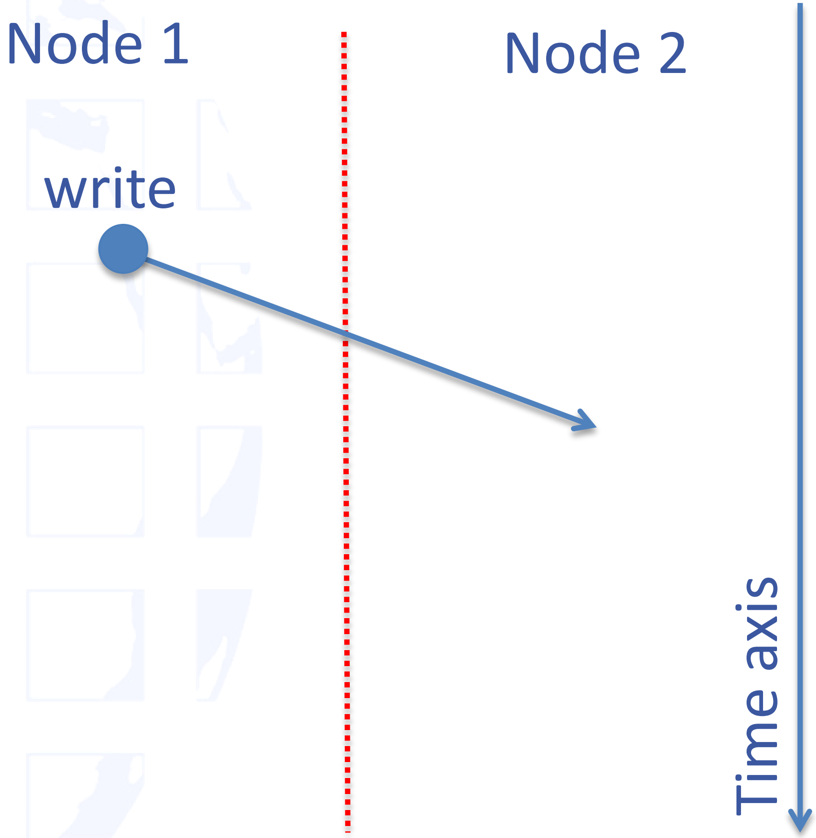


Global
Address Space
Programming Interface
GASPI



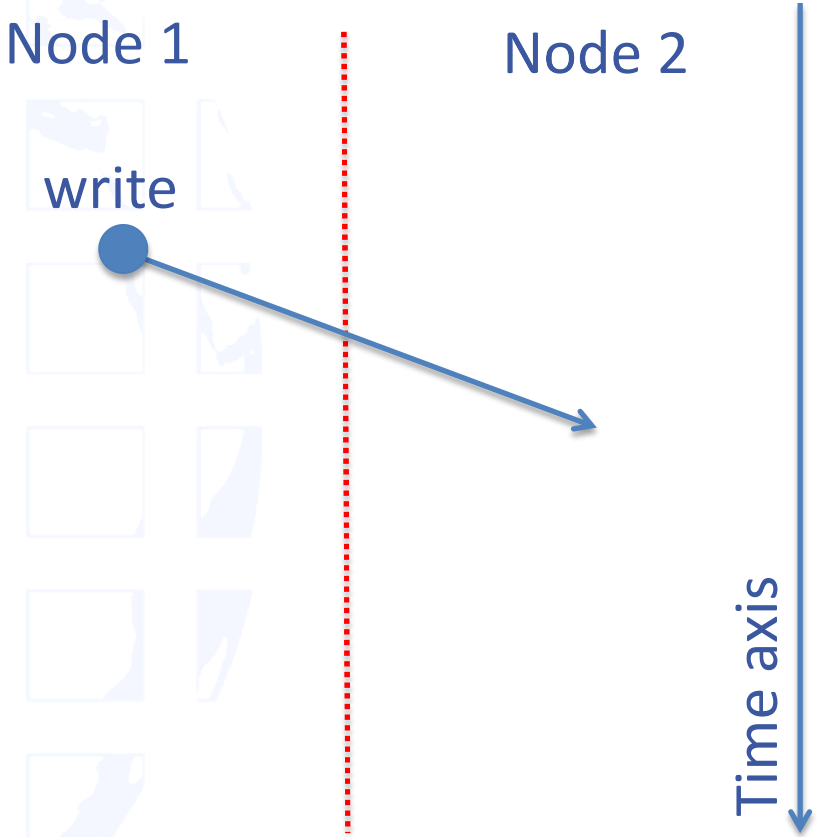
Concepts: one-sided Communication

- One-sided operations between parallel processes include remote reads and writes.
- Data can be accessed without participation of the remote site.
- One-sided communication is non-blocking: communication is triggered but may not be finished



Concepts: one-sided Communication

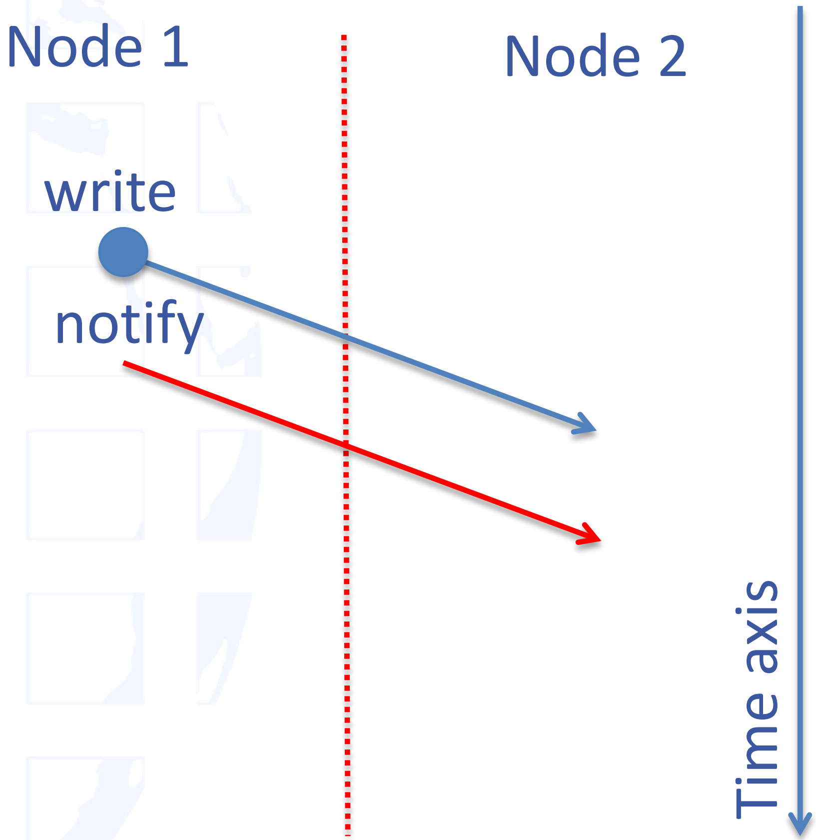
- Node 2 has not participated, it does not know that communication has started



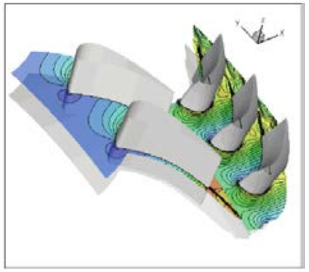
Global
Address Space
Programming Interface
GASPI

Concepts: Synchronisation with Notifications

- Node 2 has not participated, it does not know that communication has started
- It has to be notified.

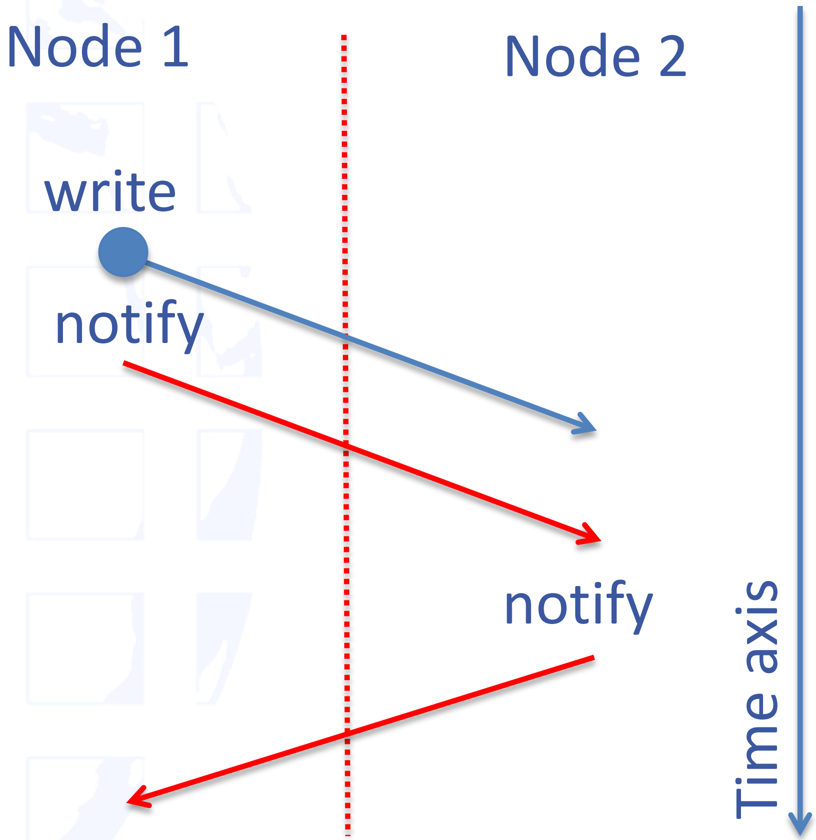


Global
Address Space
Programming Interface
GASPI



Concepts: Synchronisation with Notifications

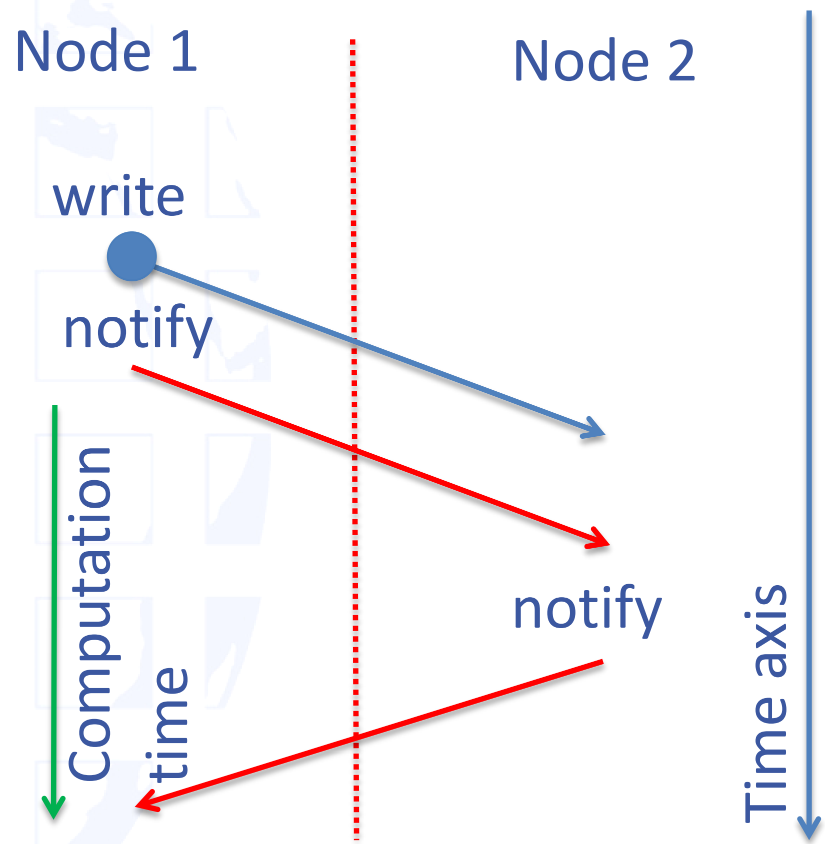
- Node 2 has not participated, it does not know that communication has started
- It has to be notified for data movement completion.
- Node 1 does not know if the write has finished.
- If it needs to know, it also has to be notified



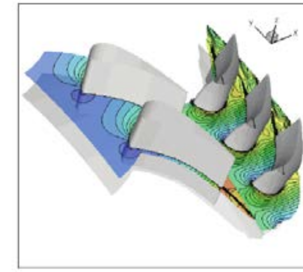
Global
Address Space
Programming Interface
GASPI

Concepts: overlap of Communication and Computation

- Due to the non-blocking nature of the call Node 1 has gained some computation time which it can use
- Communication and computation happen in parallel
- Communication latency is hidden

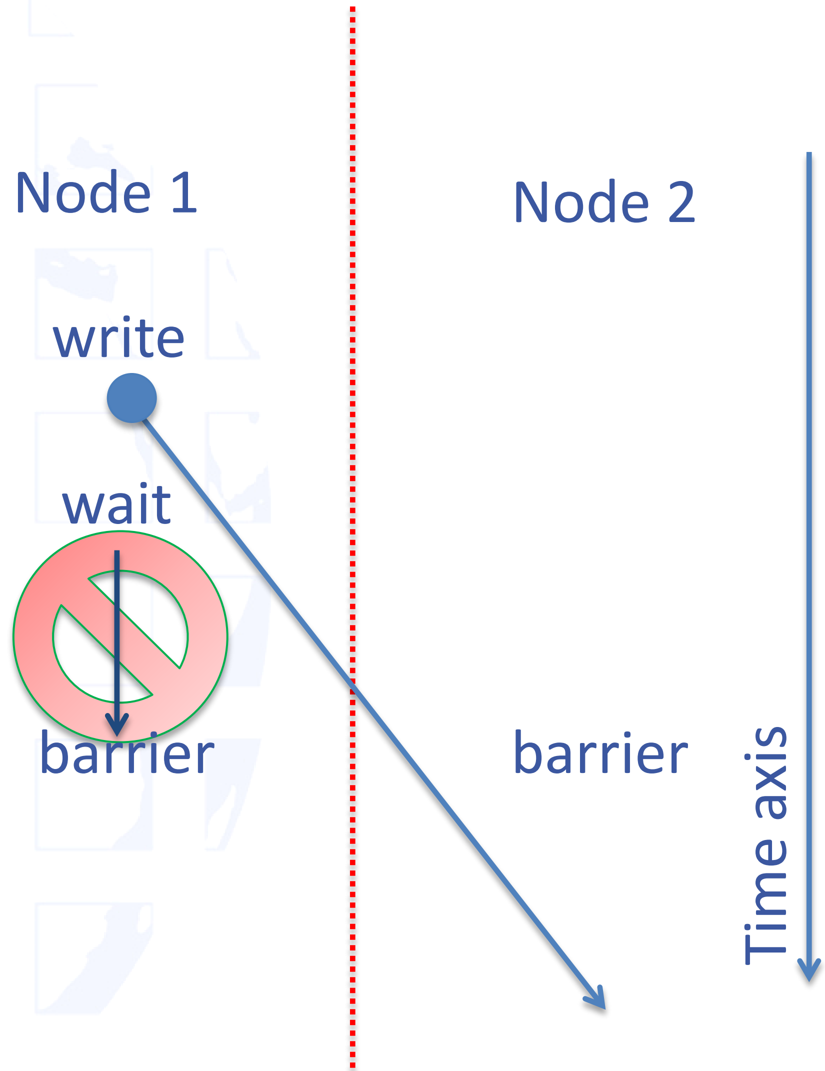


Global
Address Space
Programming Interface
GASPI

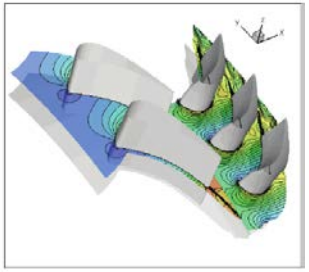


Concepts: Warning!

- Data synchronisation by wait + barrier does not work!
- Wait does wait on local queue on Node 1, does not know about write in Node 2, barrier() has no relation with communication
- Data synchronization only by notifications

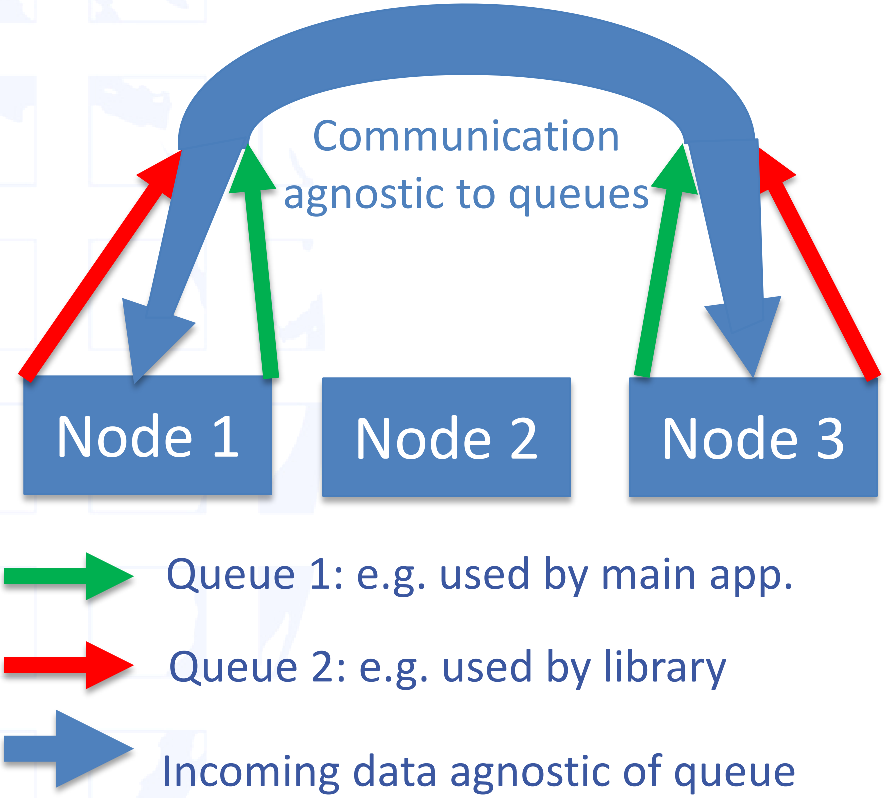


Global
Address Space
Programming Interface
GASPI



Concepts: Communication Queues

- Communication requests are posted to queues
- Queues are a local concept!
- Used to separate concerns between different parts of the applications
- Queues are used in order to establish the synchronization context.



Global
Address Space
Programming Interface
GASPI

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

```
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

www.gaspi.de

Execution Model

Global
Address Space
Programming Interface
GASPI

GASPI Execution 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

Global
Address Space
Programming Interface
GASPI

GASPI Return Values

- Procedure return values:
  - GASPI_SUCCESS
    - designated operation successfully completed
  - GASPI_TIMEOUT
    - designated operation could not be finished in the given time
    - not necessarily an error
    - the procedure has to be invoked subsequently in order to fully complete the designated operation
  - GASPI_QUEUE_FULL
    - Request could not be posted to queue. End of queue has been reached, change queue or wait
  - GASPI_ERROR
    - designated operation failed -> check error vector
- Advice: Always check return value !

Global
Address Space
Programming Interface
GASPI

success_or_die.h

```
#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)             \
  {                                   \
      printf ("Error: '%s' [%s:%i]: %i\n", #f, __FILE__, __LINE__, r);\
      exit (EXIT_FAILURE);            \
   }                                  \
} while (0)

#endif
```

Global
Address Space
Programming Interface
GASPI

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

Global
Address Space
Programming Interface
GASPI

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 Startup

- gaspi_run

```
Usage:
gaspi_run –m <machinefile>[OPTIONS] <path2bin>

Available options:
   -b <binary file> Use a different binary for
                    master
   -N               Enable NUMA for procs on same
                    node
   -n <procs>       start as many <procs> from
                    machinefile
   -d               Run with gdb on master node
```

GASPI Startup

- aprun

```
Usage:
aprun –n <procs> -d <threadsperproc> <path2bin>

Available options:
   -n               Number of processes to start
   -d               Number of threads per process
```

Build a GASPI program

- module load gpi2/1.3.0
- module swap PrgEnv-cray PrgEnv-gnu
- link the library
  - GPI2 for production
  - GPI2-dbg for development
- GPI2-dbg has several consistency checks -> more useful error messages

Global
Address Space
Programming Interface
GASPI

Hello world – Hands on

- Write a GASPI „Hello World“ program which outputs

```
Hello world from rank xxx of yyy
```

  - Use hands_on/helloworld.c as starting point
  - Use SUCCESS_OR_DIE macro to check for return values
  - Use the debug library (libGPI2-dbg.a)

- Execute the Hello World program

Global
Address Space
Programming Interface
GASPI

GASPI „hello world“

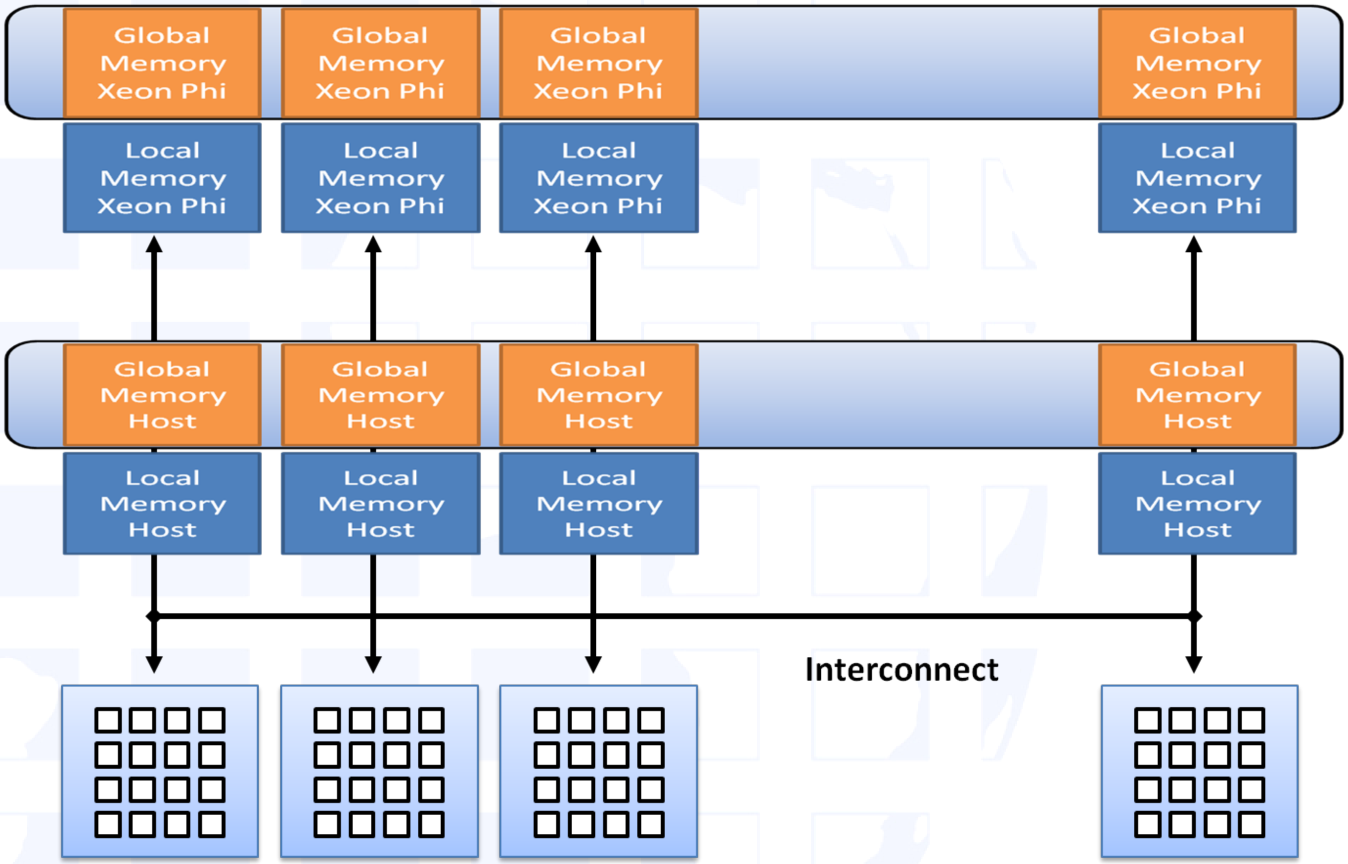
```
#include "success_or_die.h“
#include <GASPI.h>
#include <stdlib.h>

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) );

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

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

Global
Address Space
Programming Interface
GASPI

Memory Segments

Global
Address Space
Programming Interface
GASPI

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

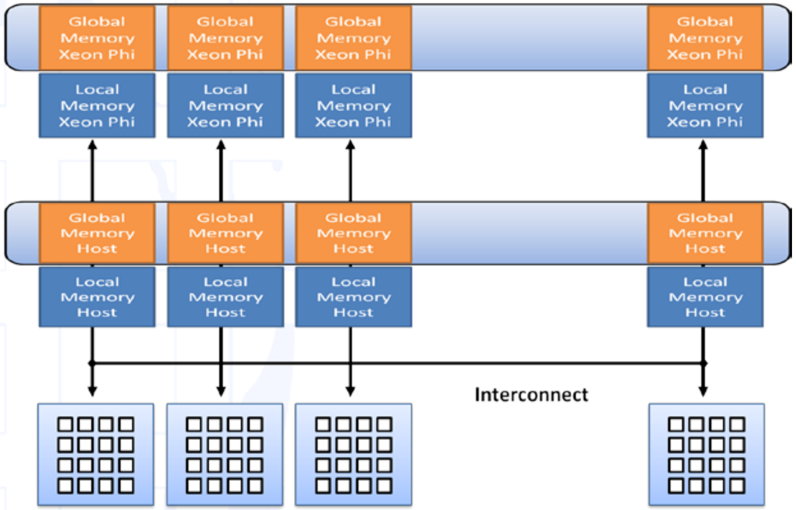
Global
Address Space
Programming Interface
GASPI

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 with given Buffer

- gaspi_segment_bind

```
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
  )
```

- Binds a buffer to a particular segment
- Same capabilities as allocated/created segment
- Locally accessible (requires gaspi_segment_register)

GASPI Segment with given Buffer

- gaspi_segment_use

```
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
  )
```

- Equivalent to

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

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

  GASPI_BARRIER (group, timeout);
}
```

Global
Address Space
Programming Interface
GASPI

GASPI Segment Deletion

- gaspi_segment_delete

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

  - Free segment memory

Global
Address Space
Programming Interface
GASPI

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 )
```

GASPI Segment Utils

- gaspi_segment_max

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

- Maximum number of segments
- Defines range of allowed segment IDs [0,segment_max - 1)

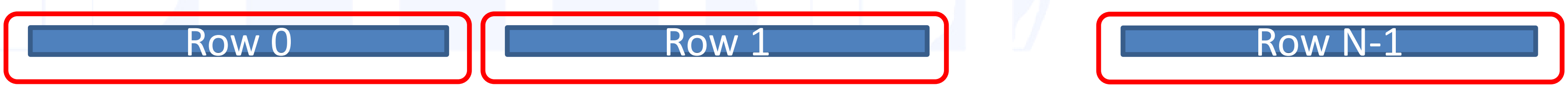
Global Address Space Programming Interface GASPI

Using Segments – Hands on

- Write a GASPI program which stores a NxM matrix in a distributed way: 1 row per process

0	1	…	M-1
M	M+1	…	2M-1
(N-1)M	(N-1)M+1	…	NM-1

  - Create a segment
  - Initialize the segment



  - output the result

Global
Address Space
Programming Interface
GASPI

Using Segments (I)

```
// 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 = VLEN * sizeof (double);

    SUCCESS_OR_DIE ( gaspi_segment_create ( segment_id, segment_size
                                          , GASPI_GROUP_ALL, GASPI_BLOCK
                                          , GASPI_MEM_UNINITIALIZED ) );
```

Global
Address Space
Programming Interface
GASPI

Using Segments (II)

```
    gaspi_pointer_t array;
    SUCCESS_OR_DIE( gaspi_segment_ptr (segment_id, &array) );

    for (int j = 0; j < VLEN; ++j)
   {
        ((double *)array )[j]= (double)( iProc * VLEN + j );
         printf( "rank %d elem %d: %f \n„
                      , iProc,j,( (double *)array )[j] );
   }

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

Global
Address Space
Programming Interface
GASPI

One-sided Communication

Global
Address Space
Programming Interface
GASPI

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
  - Asynchronous execution, thin abstraction of HW queues.

Global
Address Space
Programming Interface
GASPI

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

Queues (III)

- Queues have a finite capacity
- Queues are not automatically flushed
  - Maximize time between posting the last request and flushing the queue (qwait)
- Return value GASPI_QUEUE_FULL indicates full queue.

GASPI Queue Utils

- gaspi_queue_size

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

- gaspi_queue_size_max

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

GASPI Queue Utils

- gaspi_queue_num

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

- gaspi_queue_max

```
gaspi_return_t
gaspi_queue_max ( gaspi_number_t queue_max )
```

Global
Address Space
Programming Interface
GASPI

GASPI Queue Utils

- gaspi_queue_create

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

- gaspi_queue_delete

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

Global
Address Space
Programming Interface
GASPI

write_and_wait

- serial wait on queue
- sanity checks

```
void
write_and_wait ( gaspi_segment_id_t const segment_id_local
               , gaspi_offset_t const offset_local
               , gaspi_rank_t const rank
               , gaspi_segment_id_t const segment_id_remote
               , gaspi_offset_t const offset_remote
               , gaspi_size_t const size
               , gaspi_queue_id_t const queue
               )
{
  gaspi_timeout_t const timeout = GASPI_BLOCK;
  gaspi_return_t ret;

  /* write, wait if required and re-submit */
  while ((ret = ( gaspi_write( segment_id_local, offset_local, rank,
                               segment_id_remote, offset_remote, size,
                               queue, timeout)
          )) == GASPI_QUEUE_FULL)
    {
      SUCCESS_OR_DIE (gaspi_wait (queue,
                                  GASPI_BLOCK));
    }
  ASSERT (ret == GASPI_SUCCESS);
}
```

Global
Address Space
Programming Interface
GASPI

write_notify_and_cycle

- cycle through queues
- sanity checks

```
void
write_notify_and_cycle ( gaspi_segment_id_t const segment_id_local
                       , gaspi_offset_t const offset_local
                       , gaspi_rank_t const rank
                       , gaspi_segment_id_t const segment_id_remote
                       , gaspi_offset_t const offset_remote
                       , gaspi_size_t const size
                       , gaspi_notification_id_t const notification_id
                       , gaspi_notification_t const notification_value
                       )
{
  gaspi_number_t queue_num;
  SUCCESS_OR_DIE(gaspi_queue_num (&queue_num));

  gaspi_timeout_t const timeout = GASPI_BLOCK;
  gaspi_return_t ret;


  /* write, cycle if required and re-submit */
  while ((ret = ( gaspi_write_notify( segment_id_local, offset_local, rank,
                                      segment_id_remote, offset_remote, size,
                                      notification_id, notification_value,
                                      my_queue, timeout)
                )) == GASPI_QUEUE_FULL)
    {
      my_queue = (my_queue + 1) % queue_num;
      SUCCESS_OR_DIE (gaspi_wait (my_queue,
                                  GASPI_BLOCK));
    }
  ASSERT (ret == GASPI_SUCCESS);
```

Global
Address Space
Programming Interface
GASPI

wait_for_flush_queues

- flush all queues

```
void
wait_for_flush_queues()
{
  gaspi_number_t queue_num;
  SUCCESS_OR_DIE(gaspi_queue_num (&queue_num));

  gaspi_queue_id_t queue = 0;

  /* cycle all queues and wait */
  while (queue < queue_num)
    {
      SUCCESS_OR_DIE(gaspi_wait ( queue,
                                  GASPI_BLOCK));
      ++queue;
    }
}
```

Data Synchronization By Notification

- 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 notification mechanism

Global
Address Space
Programming Interface
GASPI

GASPI Notification Mechanism

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

Global
Address Space
Programming Interface
GASPI

GASPI Notification Mechanism

- 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 Notification Mechanism

- 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 Notification Mechanism

- 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

Global
Address Space
Programming Interface
GASPI

wait_or_die

- Wait for a given notification and reset
- Sanity checks

```
#include "waitsome.h"

#include "assert.h"
#include "success_or_die.h"

void wait_or_die
 ( gaspi_segment_id_t segment_id
 , gaspi_notification_id_t notification_id
 , gaspi_notification_t expected
 )
{
 gaspi_notification_id_t id;

 SUCCESS_OR_DIE
  (gaspi_notify_waitsome (segment_id, notification_id, 1, &id, GASPI_BLOCK));

 ASSERT (id == notification_id);

 gaspi_notification_t value;

 SUCCESS_OR_DIE (gaspi_notify_reset (segment_id, id, &value));

 ASSERT (value == expected);
}
```

Global
Address Space
Programming Interface
GASPI

test_or_die

- Test for a given notification and reset
- Sanity checks

```c
#include "assert.h"
#include "success_or_die.h"

int test_or_die
  ( gaspi_segment_id_t segment_id
  , gaspi_notification_id_t notification_id
  , gaspi_notification_t expected
  )
{
  gaspi_notification_id_t id;
  gaspi_return_t ret;

  if ( ( ret =
        gaspi_notify_waitsome (segment_id, notification_id, 1, &id, GASPI_TEST)
      ) == GASPI_SUCCESS
    )
  {
    ASSERT (id == notification_id);

    gaspi_notification_t value;

    SUCCESS_OR_DIE (gaspi_notify_reset (segment_id, id, &value));

    ASSERT (value == expected);

    return 1;
  }
  else
  {
    ASSERT (ret != GASPI_ERROR);

    return 0;
  }
}
```

Global
Address Space
Programming Interface
GASPI

Hardware Optimized

Extended One-sided Calls

- gaspi_write_notify
  - write + subsequent gaspi_notify, unordered with respect to „other“ writes.
- gaspi_write_list
  - several subsequent gaspi_writes to the same rank
- gaspi_write_list_notify
  - gaspi_write_list + subsequent gaspi_notify, non-ordered with respect to „other“ writes.
- gaspi_read_list
  - Several subsequent read from the same rank.
- gaspi_read_notify
  - read + subsequent gaspi_notify, unordered with respect to „other“ writes.

GASPI extended one-sided

- gaspi_write_notify

```
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 )
```

  – gaspi_write with subsequent gaspi_notify
  – Unordered relative to other communication (!)

GASPI extended one-sided

- gaspi_write_list

```
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 )
```

  – Several subsequent gaspi_write

Global
Address Space
Programming Interface
GASPI

GASPI extended one-sided

- gaspi_write_list_notify

```
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 )
```

  – several subsequent gaspi_write and a notification
  – Unordered relative to other communication (!)

Global
Address Space
Programming Interface
GASPI

GASPI extended one-sided

- gaspi_read_list

```
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 )
```

  – several subsequent gaspi_read

GASPI extended one-sided

- gaspi_read_notify

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

  - „gaspi_read with subsequent gaspi_notify“
  - Unordered relative to other communication (!)

Communication – Hands on

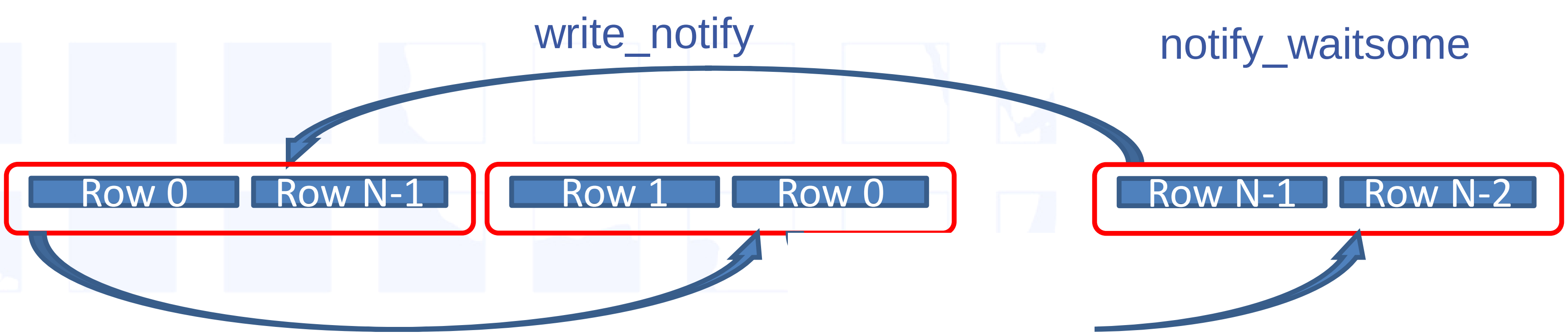
- Take your GASPI program which stores a NxM matrix in a distributed way and extend it by communication for rows

0	1	…	M-1
M	M+1	…	2M-1
(N-1)M	(N-1)M+1	…	NM-1

  - Create a segment (sufficient size for a source and target row)
  - Initialize the segment

Communication – Hands on

- Take your GASPI program which stores a NxM matrix in a distributed way and extend it by communication
  - Communicate your row to your right neighbour (periodic BC)



  - Check that the data is available
  - Output the result

Global
Address Space
Programming Interface
GASPI

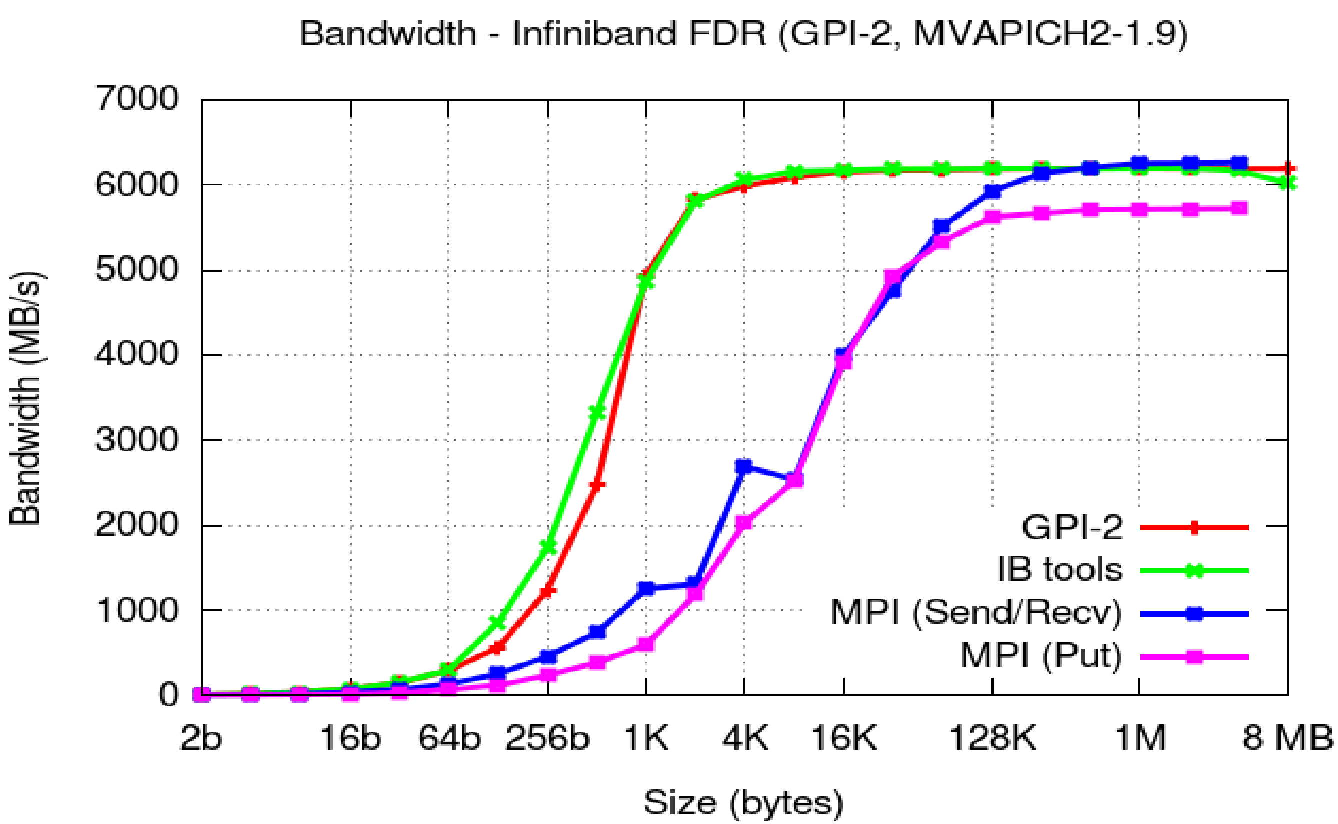
onesided.c (I)

```
// 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 ); }
```

```
/* write, cycle if required and re-submit */
 while ((ret = ( gaspi_write_notify( segment_id_local, offset_local, rank,
                                      segment_id_remote, offset_remote, size,
                                      notification_id, notification_value,
                                      my_queue, timeout)
                )) == GASPI_QUEUE_FULL) {
     my_queue = (my_queue + 1) % queue_num;
     SUCCESS_OR_DIE (gaspi_wait (my_queue,
                                 GASPI_BLOCK));
 }
 ASSERT (ret == GASPI_SUCCESS);
```

```
gaspi_notification_id_t data_available = 0;
gaspi_offset_t loc_off = 0;
gaspi_offset_t rem_off = VLEN * sizeof (double);
write_notify_and_cycle ( segment_id
                       , loc_off
                       , RIGHT (iProc, nProc)
                       , segment_id
                       , rem_off
                       , VLEN * sizeof (double)
                       , data_available
                       , 1 + iProc
                       );
wait_or_die (segment_id, data_available, 1 + LEFT (iProc, nProc) );
for (int j = 0; j < VLEN; ++j)
{ 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;}
```

GPI 2.0 - Bandwidth



Global
Address Space
Programming Interface
GASPI

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

Global
Address Space
Programming Interface
GASPI

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 )
```

Global
Address Space
Programming Interface
GASPI

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
    - „Passive“ Active Messages (see advanced tutorial code)
      - GASPI Swiss Army Knife.

Global
Address Space
Programming Interface
GASPI

Passive Communication Functions (III)

- Example: Negotiate offsets for alltoallV communication
  - Set local send offsets, local receive offsets and remote receive offsets.
  - Use passive communication for serializing incoming traffic in order to determine linear alltoallV workarrays.
  - Use passive communication to trigger remote printing of received data.

Global
Address Space
Programming Interface
GASPI

Passive Communication Functions (IV)

```
void *handle_passive(void *arg)
{
  gaspi_pointer_t _vptr;
  SUCCESS_OR_DIE(gaspi_segment_ptr(passive_segment, &_vptr));
  const gaspi_offset_t passive_offset = sizeof(packet);
  while(1)
    {
      gaspi_rank_t sender;
      SUCCESS_OR_DIE(gaspi_passive_receive(passive_segment
                                           , passive_offset
                                           , &sender
                                           , sizeof(packet)
                                           , GASPI_BLOCK
                                           ));
      packet *t = (packet *) ((char*)_vptr + sizeof(packet));
      return_offset(t->rank, t->len, t->offset)
    }
return NULL;
```

Global
Address Space
Programming Interface
GASPI

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

Global
Address Space
Programming Interface
GASPI

Interoperability with MPI

Global
Address Space
Programming Interface
GASPI

Interoperability with MPI

- GASPI supports interoperability with MPI in a so-called mixed-mode.
- The mixed-mode allows for
  - either entirely porting an MPI application to GASPI
  - or replacing performance-critical parts of an MPI based application with GASPI code (useful when dealing with large MPI code bases)
- Porting guides available at:

  http://www.gpi-site.com/gpi2/docs/whitepapers/

Global
Address Space
Programming Interface
GASPI

Mixing GASPI and MPI in Parallel Programs

- GASPI must be installed with MPI support, using the option
  --with-mpi <path_to_mpi_installation>
- MPI must be initialized before GASPI, as shown in the joined example
- The same command or script as the one provided by the MPI installation should be used for starting programs (mpirun or similar)
- gaspi_run should not be used!

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

int main (int argc, char *argv[])
{
  // initialize MPI and GASPI
  MPI_Init (&argc, &argv);
  gaspi_proc_init (GASPI_BLOCK);

  // Do work …

  // shutdown GASPI and MPI
  gaspi_proc_term (GASPI_BLOCK);
  MPI_Finalize();

  return 0;
}
```

Global
Address Space
Programming Interface
GASPI

GASPI Preserves the MPI Ranks

- GASPI is able to detect at runtime the MPI environment and to setup its own environment based on this
- GASPI can deliver the same information about ranks and number of processes as MPI
- This helps to preserve the application logic

```
…

int nProc_MPI, iProc_MPI;
gaspi_rank_t iProc, nProc;

MPI_Init(&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &iProc_MPI);
MPI_Comm_size (MPI_COMM_WORLD, &nProc_MPI);

SUCCESS_OR_DIE (gaspi_proc_ini(GASPI_BLOCK));
SUCCESS_OR_DIE (gaspi_proc_rank (&iProc));
SUCCESS_OR_DIE (gaspi_proc_num (&nProc));

ASSERT(iProc == iProc_MPI);
ASSERT(nProc == nProc_MPI);

…
```

Using User Provided Memory for Segments

- New feature added in version 1.3 of GASPI: a user may provide already allocated memory for segments
- Memory used in MPI communication can be used in GASPI communication
- However, the feature should be used with care because the segment creation is an expensive operation

```
//initialize and allocate memory
double *buffer = calloc ( num_elements
                        , sizeof(double)
                        );
gaspi_segment_id_t segment_id = 0;

//use the allocated buffer as underlying
//memory support for a segment
SUCCESS_OR_DIE
  ( gaspi_segment_use
  , segment_id
  , buffer
  , n*sizeof (double)
  , GASPI_GROUP_ALL
  , GASPI_BLOCK
  , 0
  );
```

Global
Address Space
Programming Interface
GASPI

Using GASPI Segment Allocated Memory in MPI Communication

```
// allgatherV
 SUCCESS_OR_DIE (gaspi_segment_create ( segment_id
              , vlen * sizeof(int), GASPI_GROUP_ALL, GASPI_BLOCK
              , GASPI_ALLOC_DEFAULT));

  gaspi_pointer_t _ptr = NULL;
  SUCCESS_OR_DIE (gaspi_segment_ptr (segment_id, &_ptr));
  int *array = (int *) _ptr;
  init_array(array, offset, size, iProc, nProc);

  MPI_Allgatherv(&array[offset[iProc]], size[iProc], MPI_INT
                , array, size, offset, MPI_INT, MPI_COMM_WORLD);
```

Mixing MPI Code with GASPI Code From a Library

- In mixed-mode, an MPI based code may call GASPI code that is embedded into a library

- The GASPI environment must be initialized and cleaned up within the calling program

```
int n, my_mpi_rank, n_mpi_procs;
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &my_mpi_rank);
MPI_Comm_size (MPI_COMM_WORLD, &n_mpi_procs);

SUCCESS_OR_DIE (gaspi_proc_init, GASPI_BLOCK);

// initialize data
// distribute data, do MPI communication
// call GPI library function for iteratively
// solving a linear system
Gaspi_Jacobi( n, n_local_rows, local_a,
          , local_b, &x, x_new, n_max_iter, tol
          );

SUCCESS_OR_DIE (gaspi_proc_term, GASPI_BLOCK);
MPI_Finalize();
```

Global
Address Space
Programming Interface
GASPI

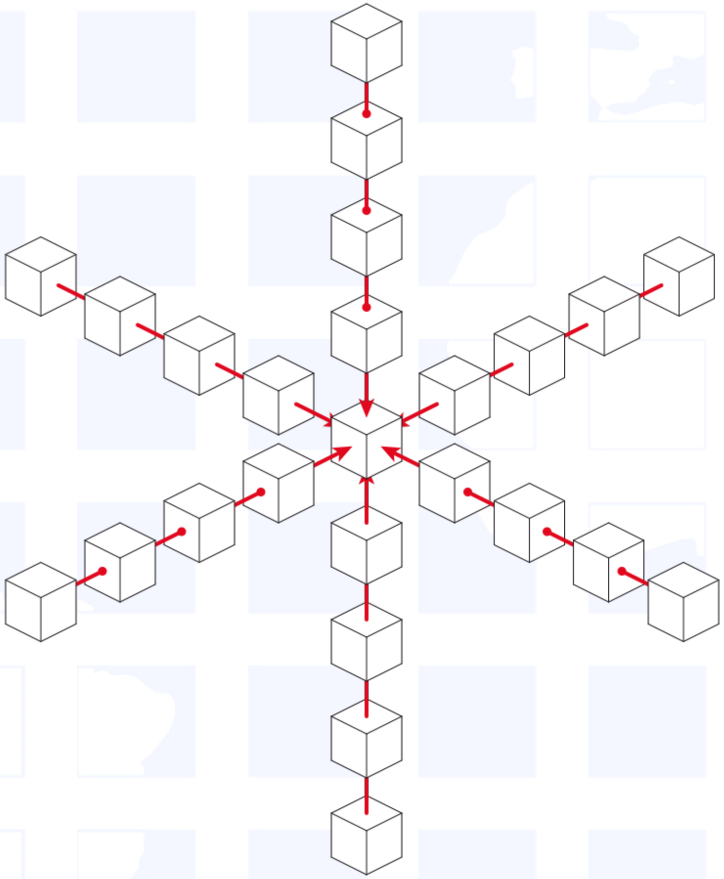
The GASPI programming model

Global
Address Space
Programming Interface
GASPI

Asynchronous execution
with maximal overlap of communication and computation

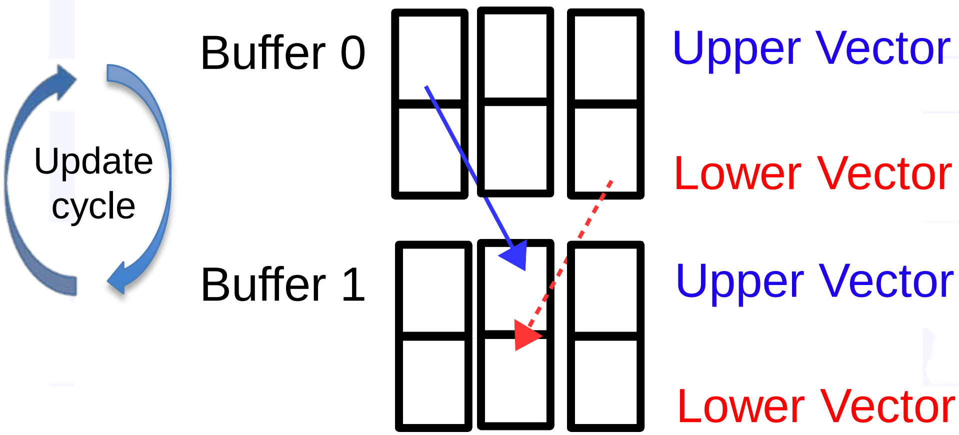
THINK PERFORMANCE

Example: Stencil applications



- Important class of algorithms
  - FD methods
  - Image processing
  - PDEs
- Iterative method
- Non-local updates
  -> data dependencies

Stencil application proxy

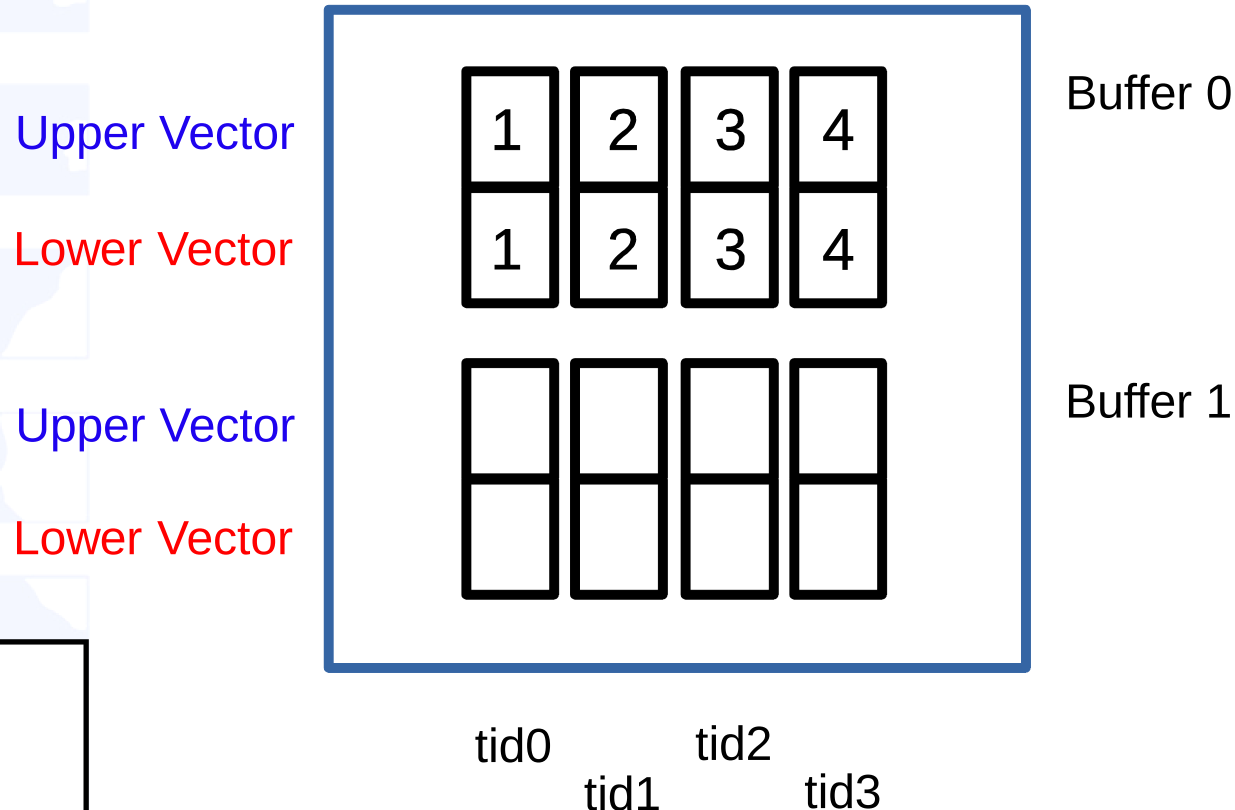


Update step:
- Update upper part
- Update lower part

- 2 buffers per element
  - Buffer 0
  - Buffer 1
- 2 vectors per buffer
  - Upper vector
  - Lower vector
- Data dependencies
  - Left element
  - Right element

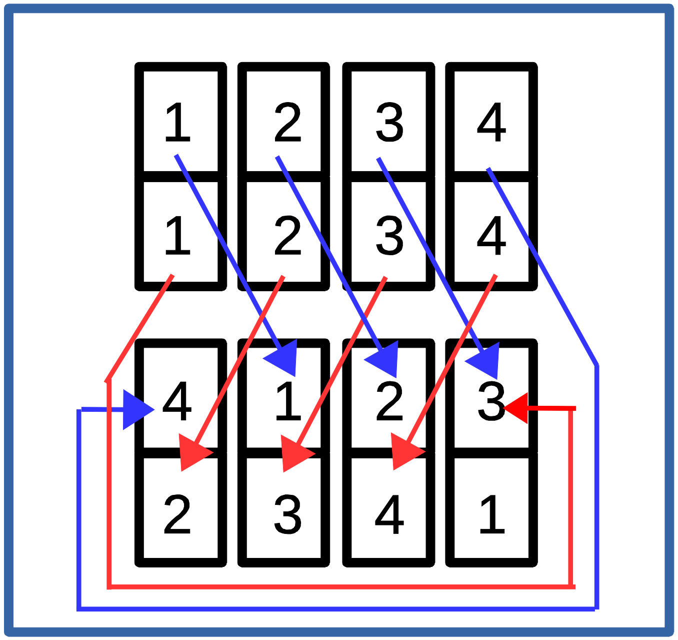
Global
Address Space
Programming Interface
GASPI

Stencil application proxy



- Nthread omp threads
- static domain decomposition / assignment
- Two buffers per thread
- Two vectors per buffer
- Vector length: nvector

Iteration 1



Upper half: move to the right

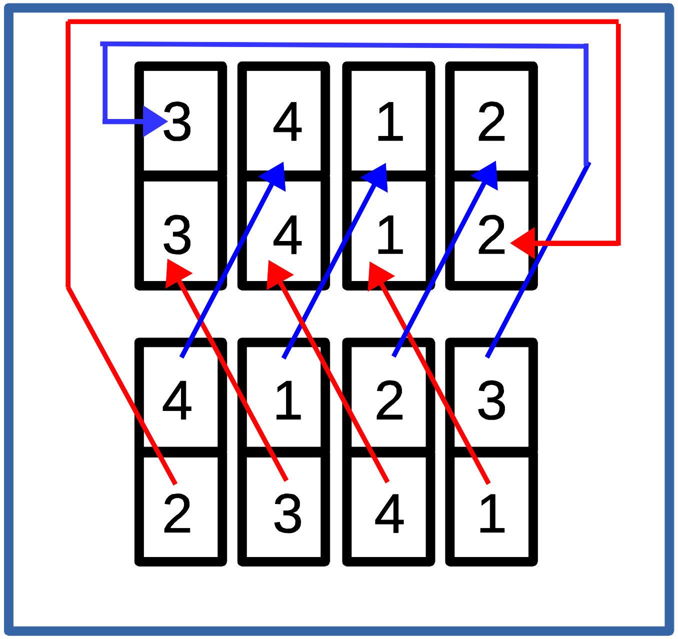
Lower half: move to the left

Periodic BC

Periodic BC

barrier

Iteration 2



Periodic BC

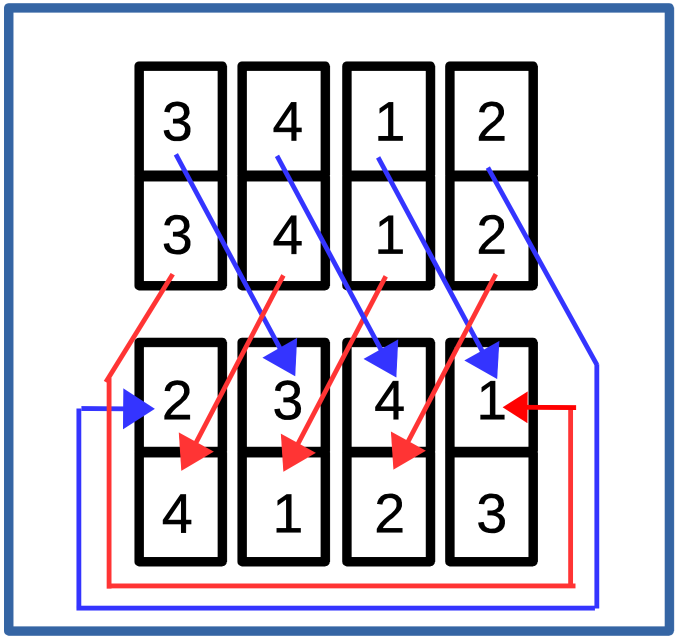
Periodic BC

Upper half: move to the right

Lower half: move to the left

barrier

Iteration 3



Upper half: move to the right

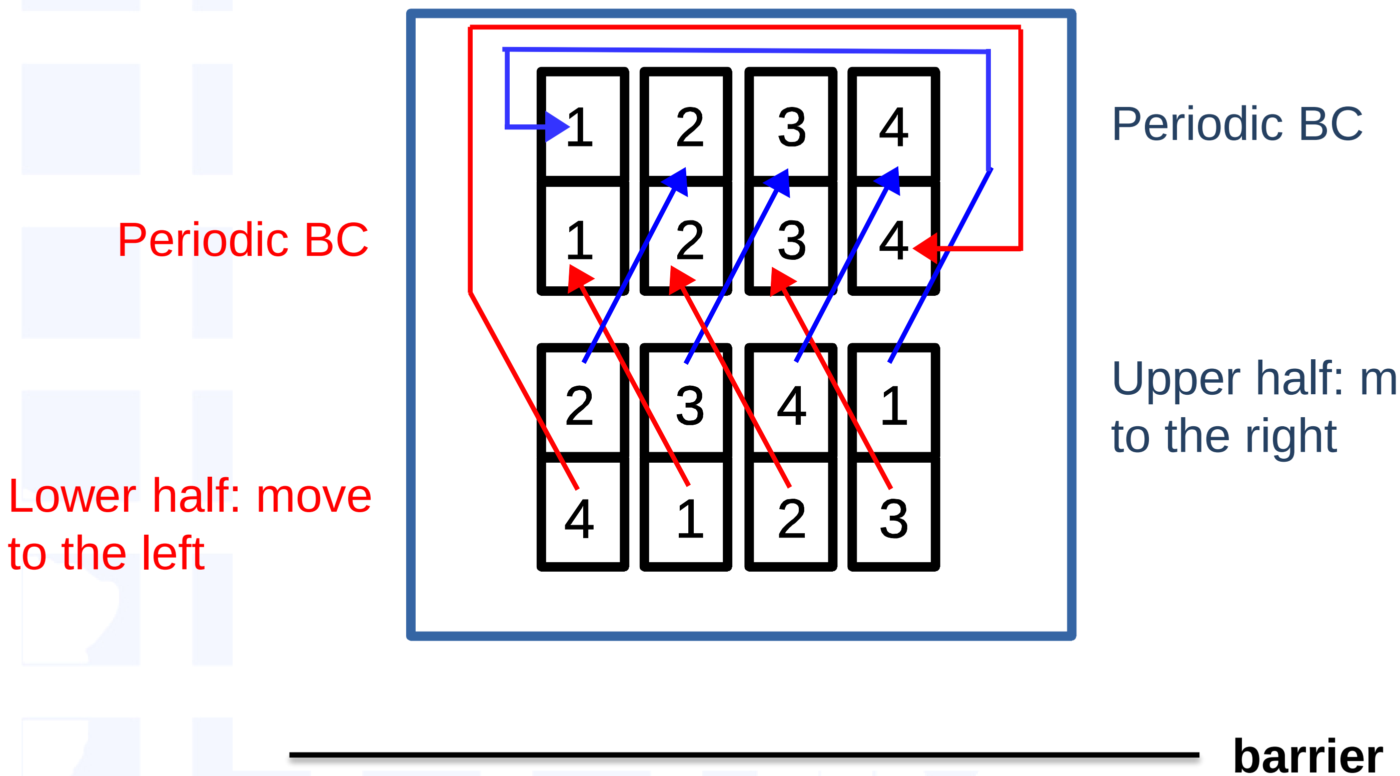
Lower half: move to the left

Periodic BC

Periodic BC

barrier

Iteration 4

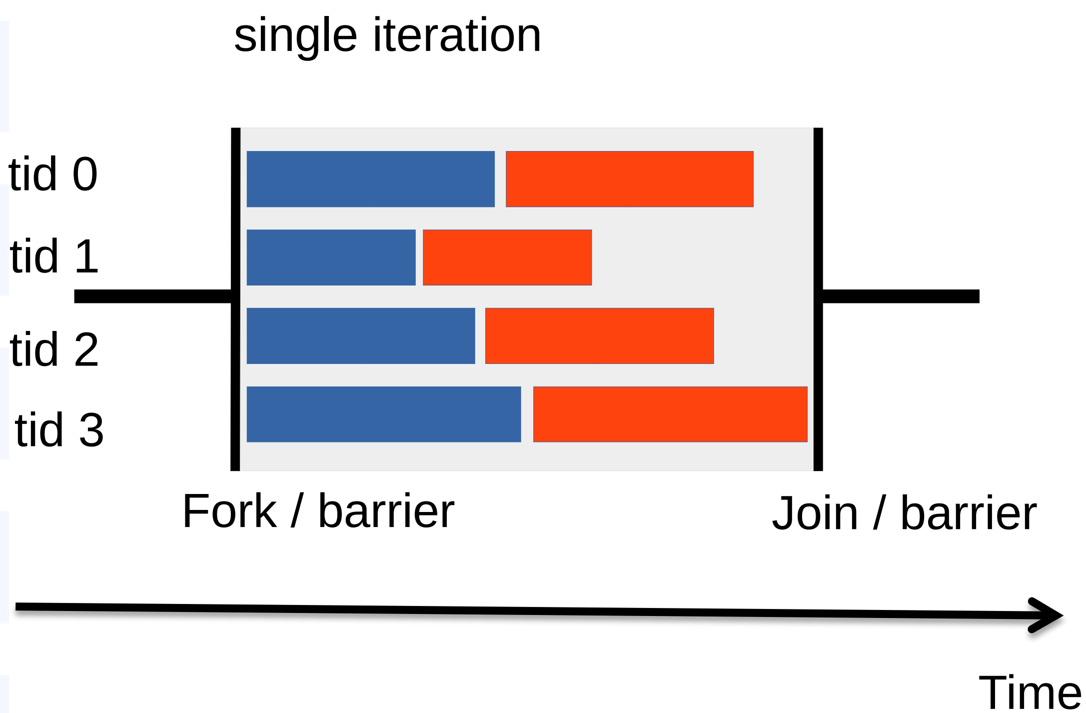

Periodic BC
Periodic BC
Upper half: move to the right
Lower half: move to the left
1
2
3
4
1
2
3
4
2
3
4
1
4
1
2
3
barrier

- Nelem many iterations:
  - Initial configuration recovered

  -> Easy to check

Global
Address Space
Programming Interface
GASPI

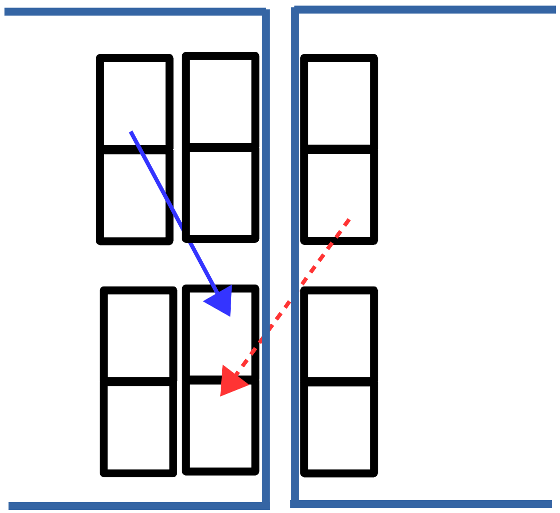
Temporal evolution



Global
Address Space
Programming Interface
GASPI

MORE THAN ONE PROCESS …

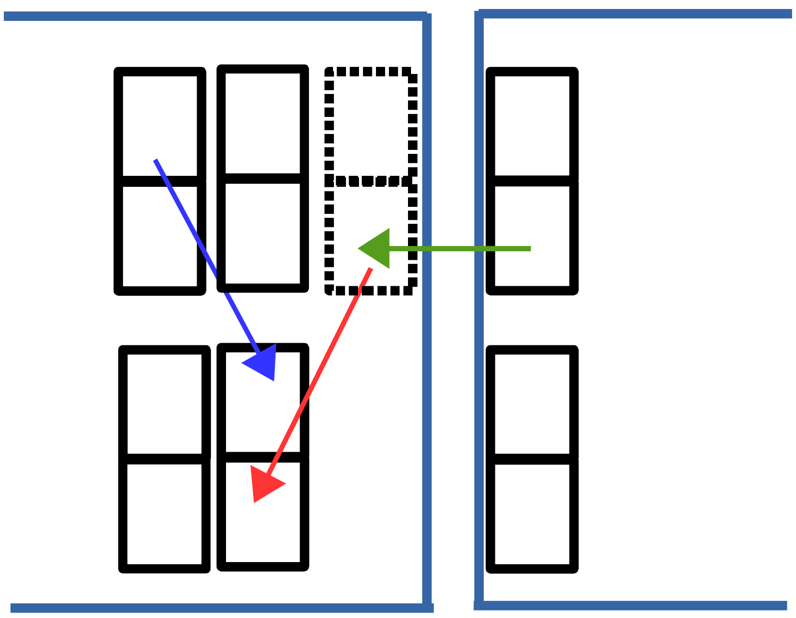
Elementary update

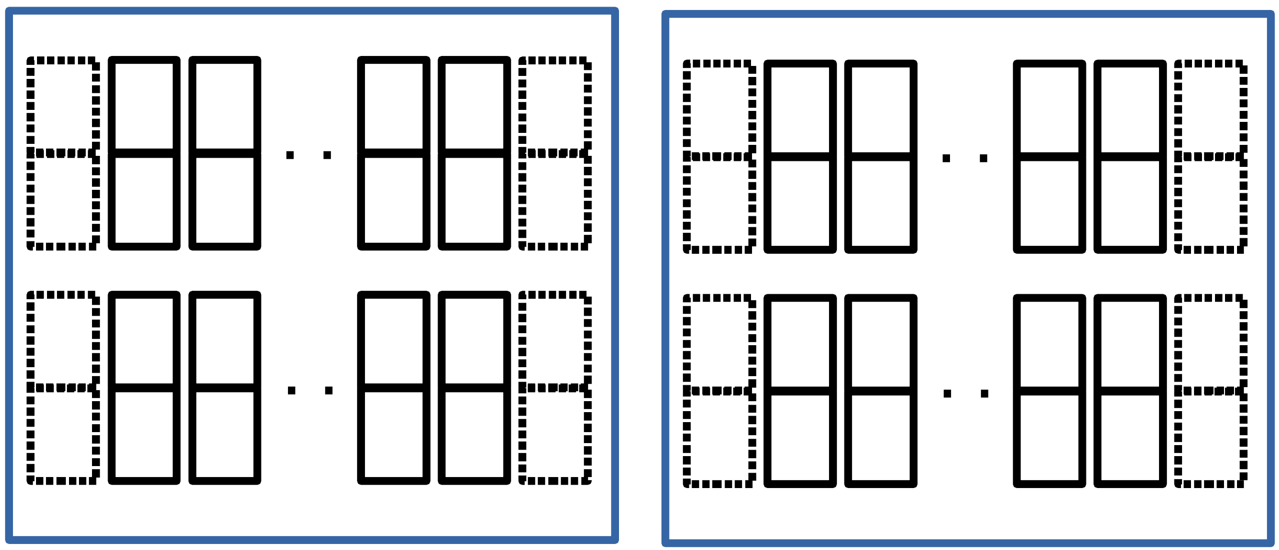


- Each process hosts some part of the information
- Part of the information is no longer directly accessible

Global
Address Space
Programming Interface
GASPI

Boundary / Halo domains





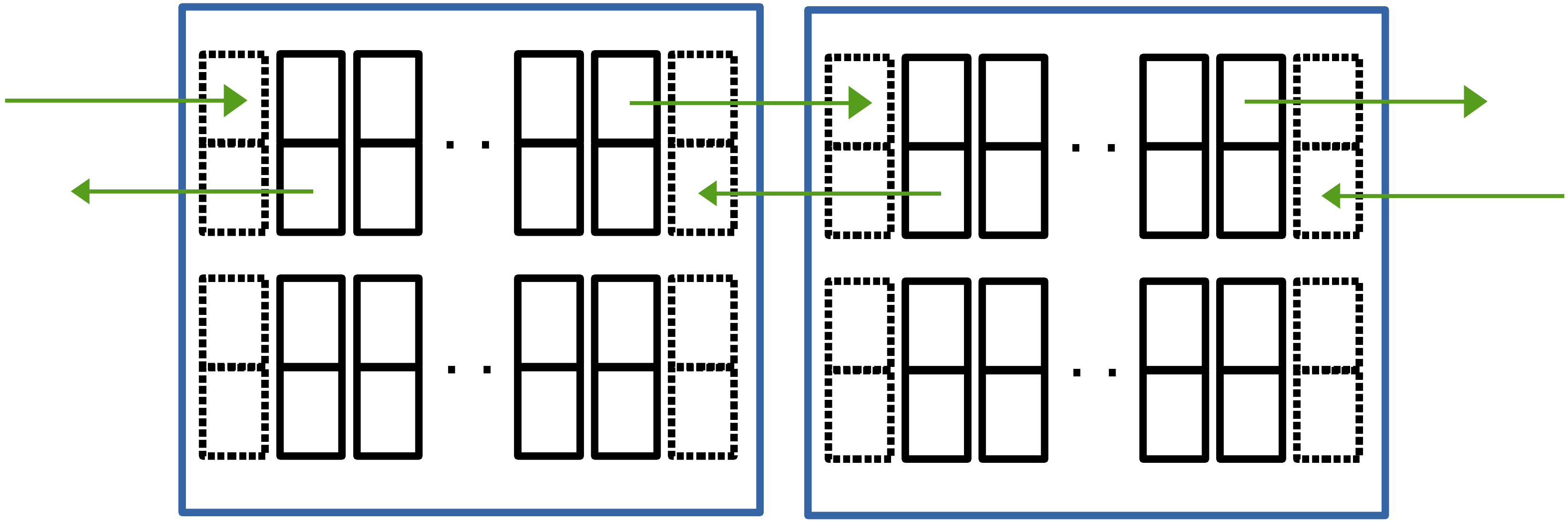
Global
Address Space
Programming Interface
GASPI

Separate communication / computation phases

BULK SYNCHRONOUS

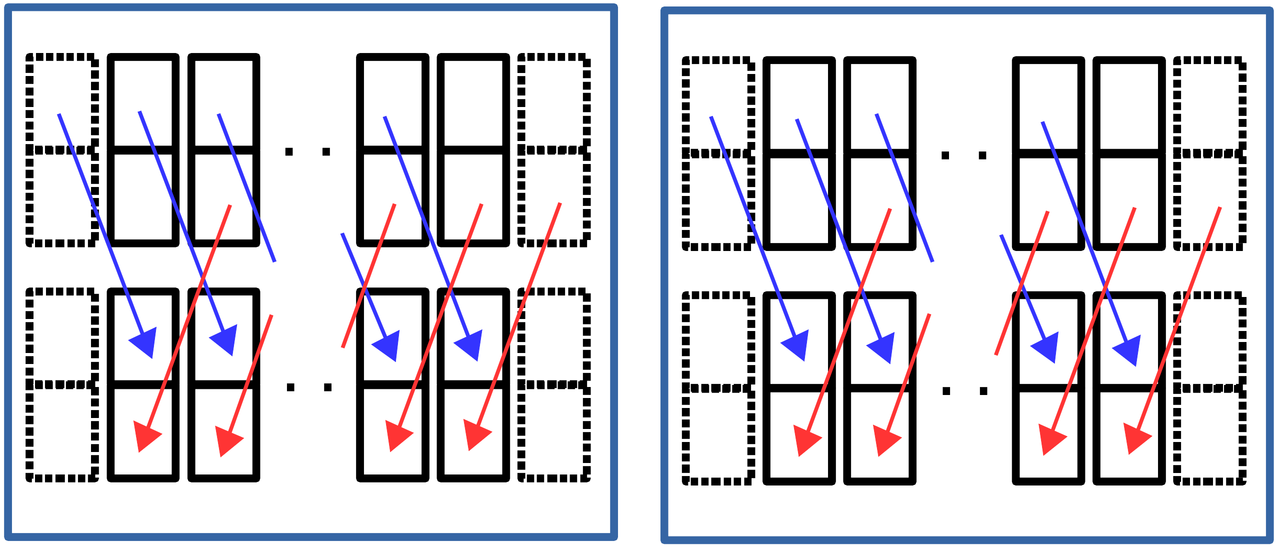
Global
Address Space
Programming Interface
GASPI

Communication phase



barrier barrier

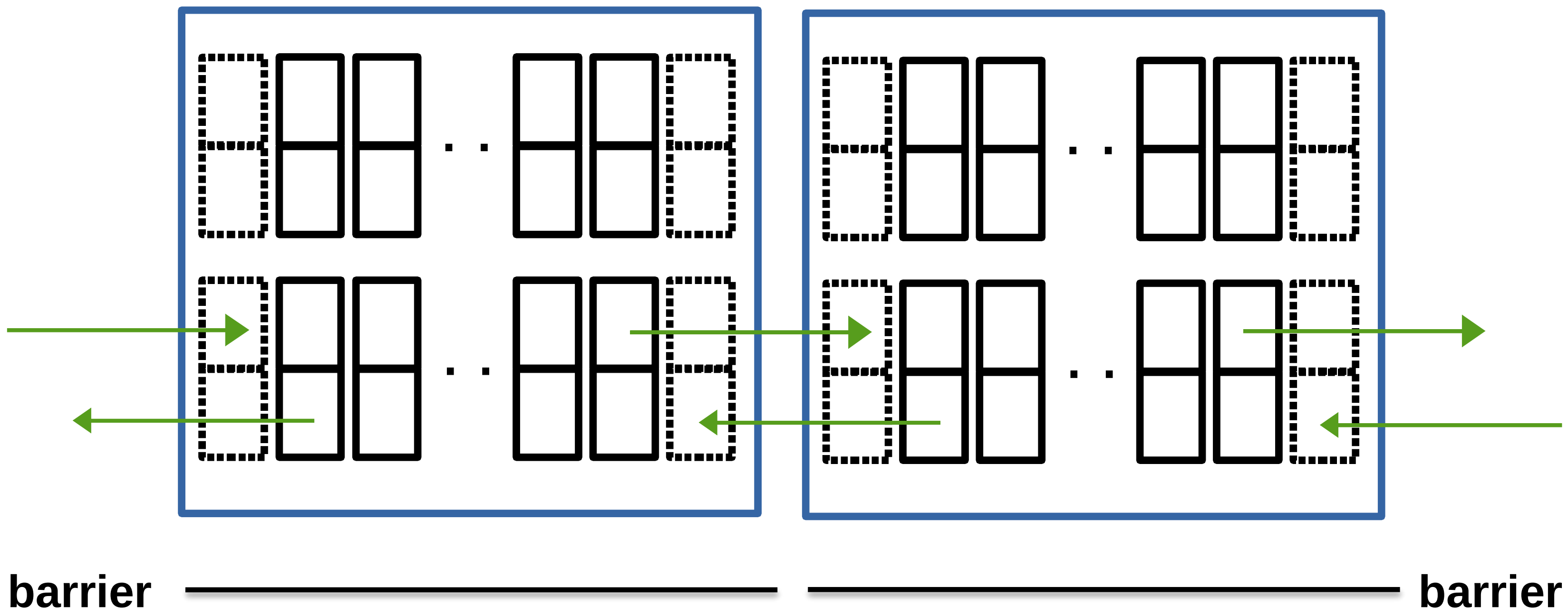
Computation phase



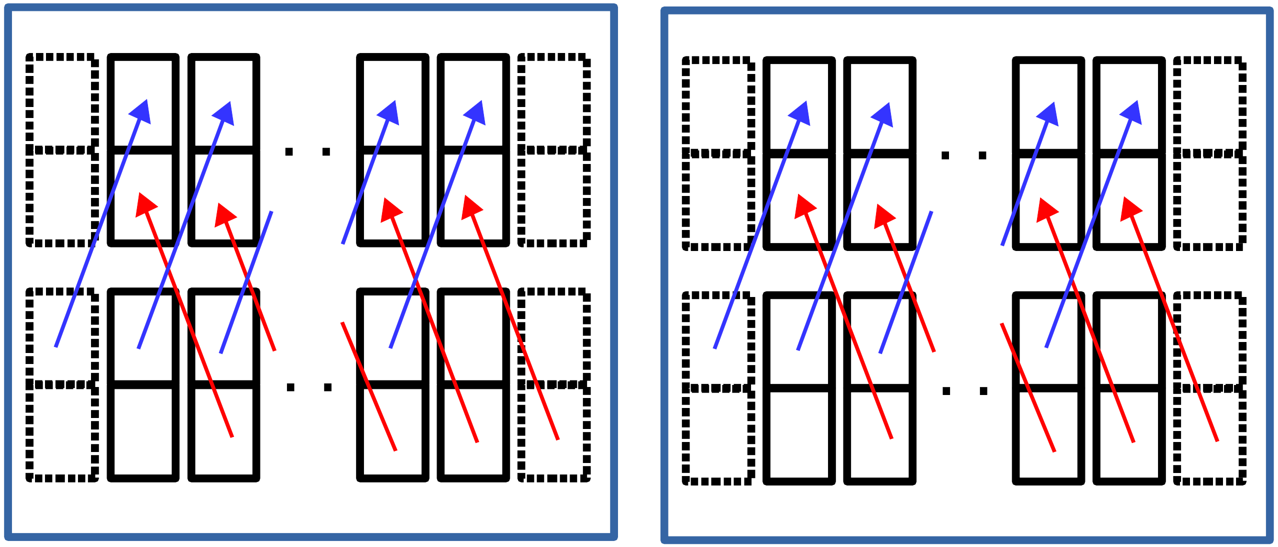
barrier ——————————————— barrier

Global
Address Space
Programming Interface
GASPI

Communication phase



Computation phase



barrier —————— —————— barrier

The GASPI Ring Exchange

- GASPI – left_right_double_buffer_funneled.c

```
 if (tid == 0) {
 // issue write
 write_notify_and_cycle
  ( .. , LEFT(iProc, nProc),., right_data_available[buffer_id], 1 + i);
 // issue write
 write_notify_and_cycle
  ( .., RIGHT(iProc, nProc),., 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;
```

Basic ingredients

EXCURSION: EFFICIENT PARALLEL EXECUTION

Global
Address Space
Programming Interface
GASPI

Efficient parallel execution

- Q: What is the measure for „efficient parallel execution“ ?
- A: Scalability

Global
Address Space
Programming Interface
GASPI

Efficient parallel execution

- Definition: $S(N_{proc}) = \dfrac{T(1)}{T(N_{proc})}$
- Interpretation:

Measure for the additional benefit generated by employing additional resources

Scalability S

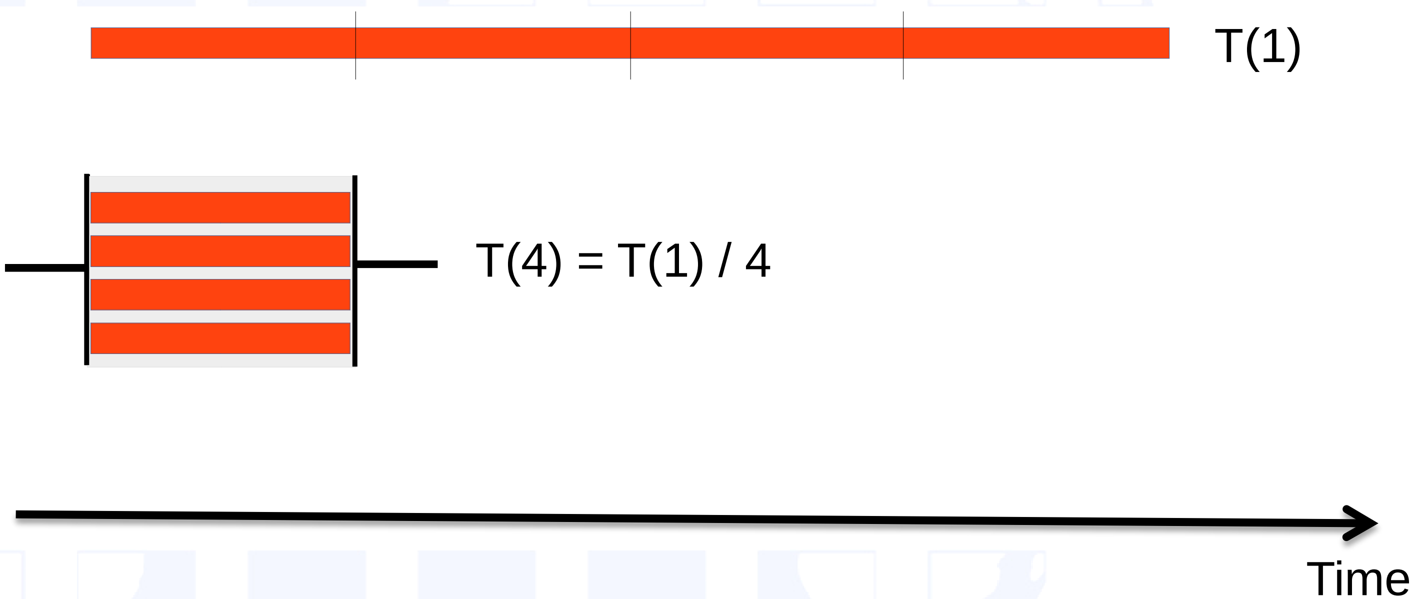
- Optimal: linear scalability, i.e.

  $T(N_{proc}) = T(1)/N_{proc}$

  -> doubling the resources implies doubling the generated benefit

Implications for parallelization

- $T(N_{proc}) := T(1)/N_{proc}$



Implications for parallelization

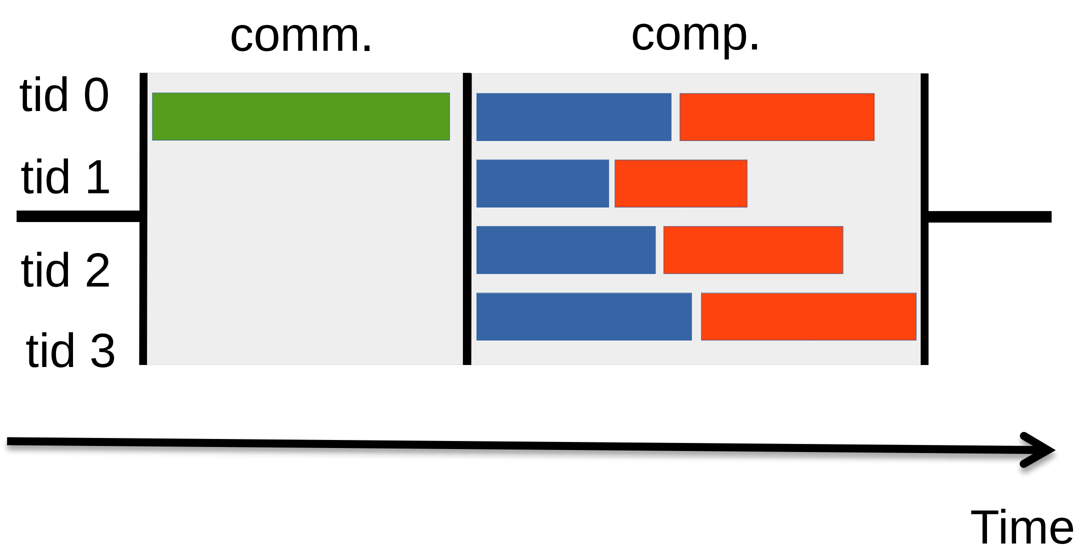
- $T(N_{proc}) := \mathrm{T}(1)/N_{proc}$
- T(1) is pure computation time, i.e.
  - communication latencies need to be completely hidden by the parallel implementation
  - Optimal load balancing is required
    - No synchronization points
      (Potential aggregation of imbalances, imbalances are per se unavoidable, e.g. OS jitter etc.)
    - Contiguous stream of computational tasks

Global
Address Space
Programming Interface
GASPI

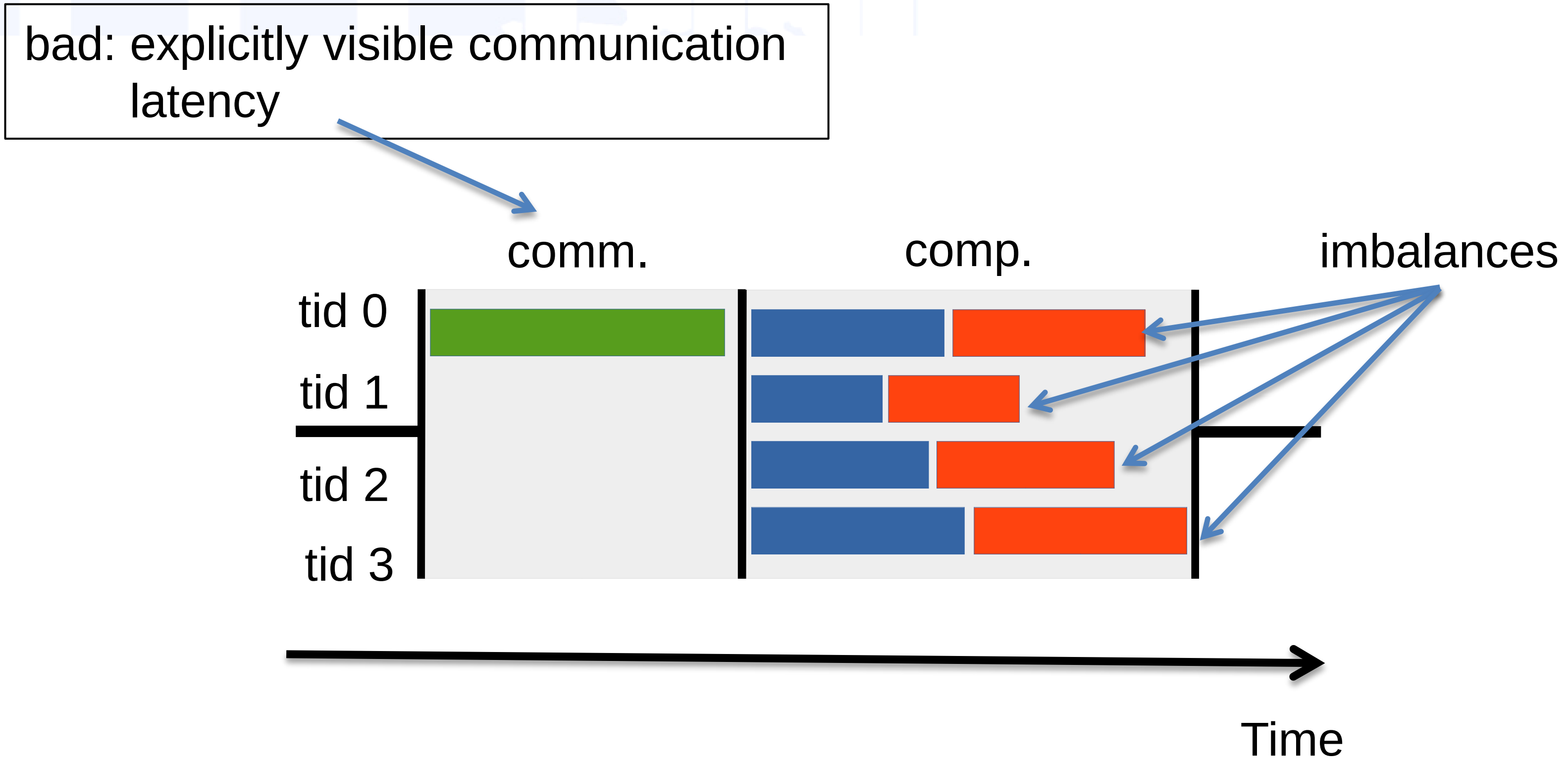
END OF EXCURSION

Global
Address Space
Programming Interface
GASPI

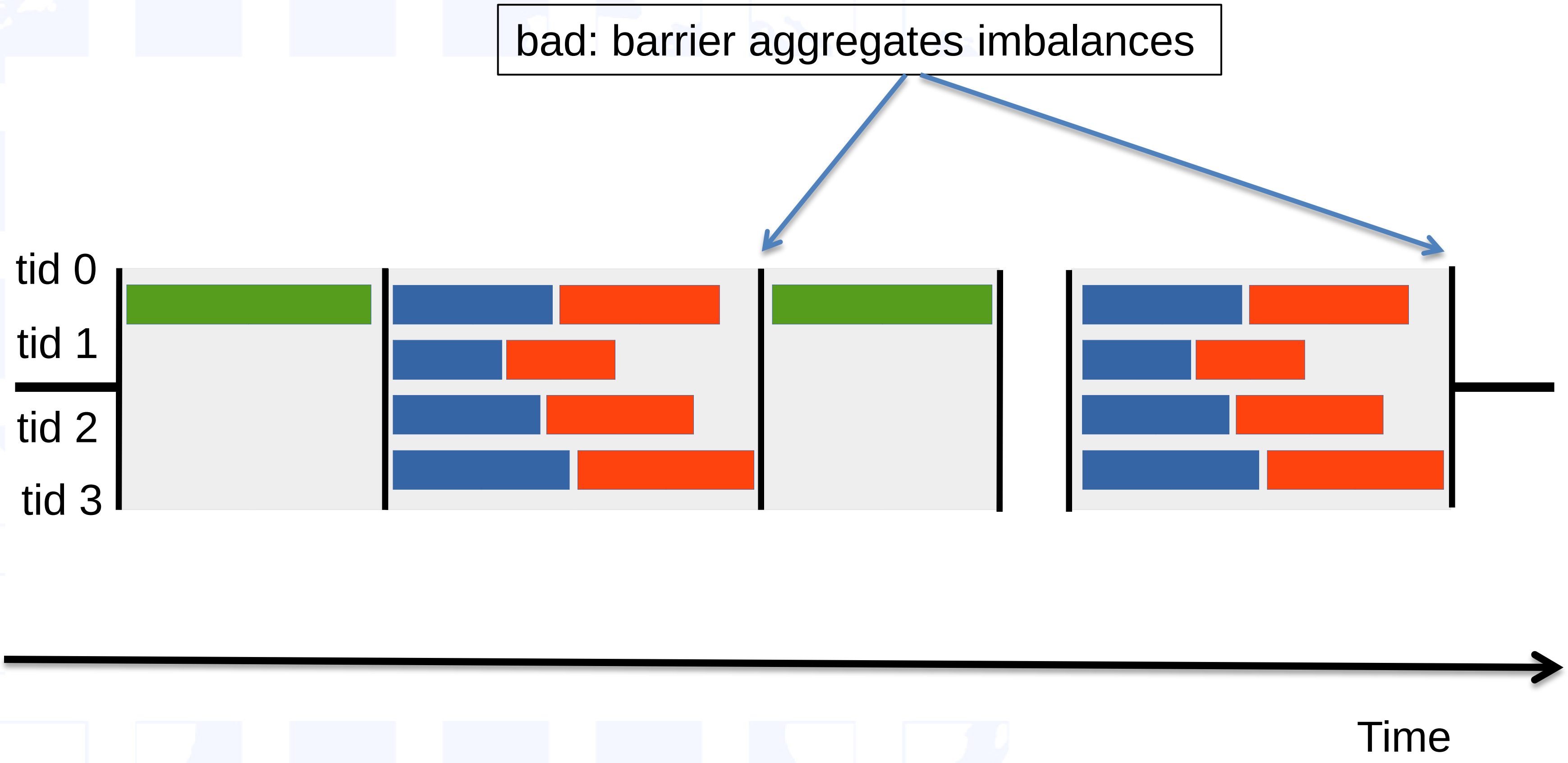
Temporal evolution: one iteration



Temporal evolution: one iteration



Temporal evolution: all iterations



Global
Address Space
Programming Interface
GASPI

Hide communication behind computation

COMMUNICATION / COMPUTATION OVERLAP

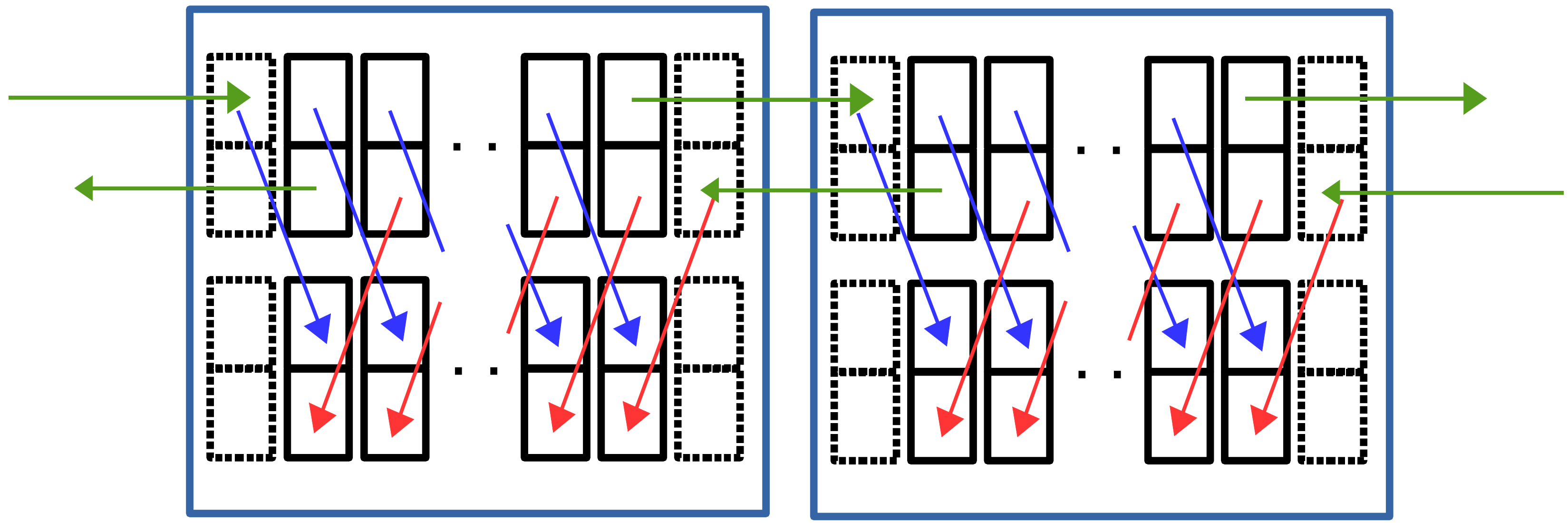
Strategy

- Hide communication latencies behind computation
- Split data into inner / boundary part
  - Inner data ⇔ no dependence on remote information
  - Boundary data ⇔ has dependence on remote information

Strategy

- Algorithmic phases:
  - Init boundary data transfer
  - Update inner data along data transfer
  - Update boundary data

Single iteration

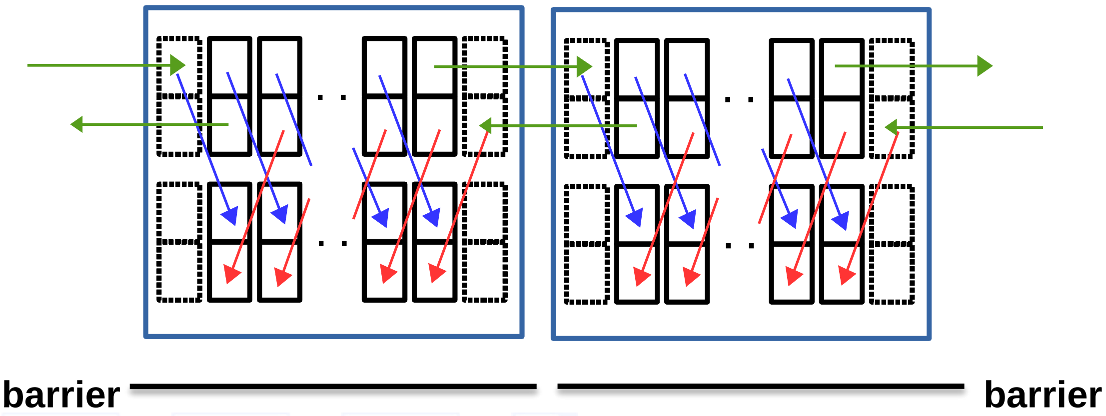


barrier barrier

Single iteration: details

Left boundary element:

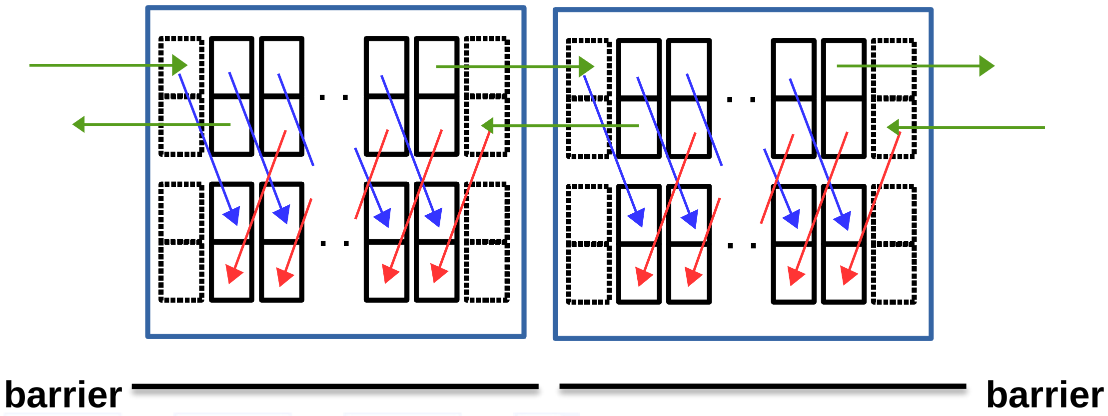
1. Initiate boundary data transfer to remote halo



Single iteration: details

Left boundary element:

1. Initiate boundary data transfer to remote halo
2. Wait for boundary data transfer to local halo completion

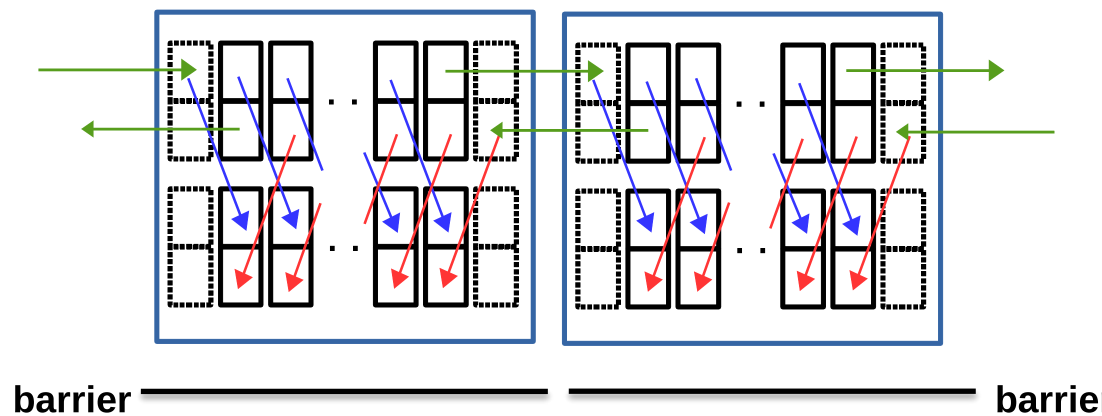


Global
Address Space
Programming Interface
GASPI

Single iteration: details

Left boundary element:

1. Initiate boundary data transfer to remote halo
2. Wait for boundary data transfer to local halo completion
3. Update vector



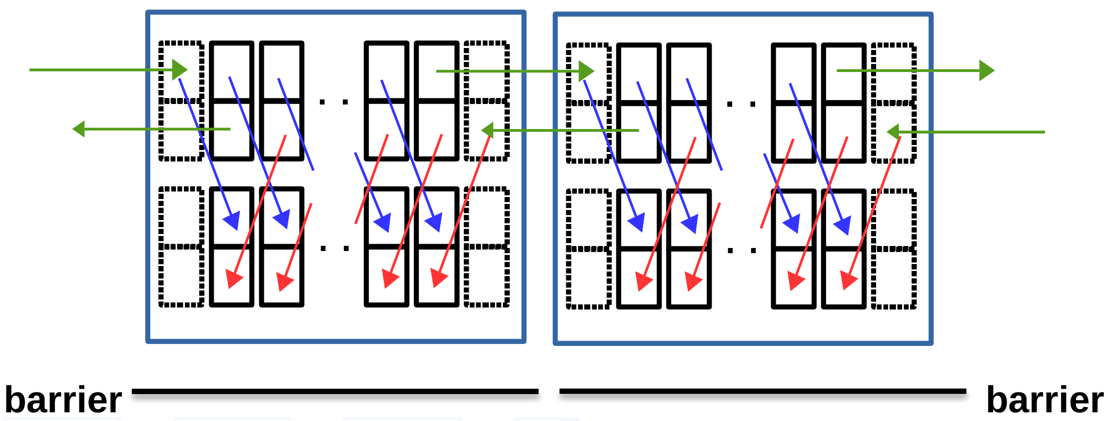
Global
Address Space
Programming Interface
GASPI

Single iteration: details

Left boundary element:

1. Initiate boundary data transfer to remote halo
2. Wait for boundary data transfer to local halo completion
3. Update vector

-> Right boundary element handled analogously



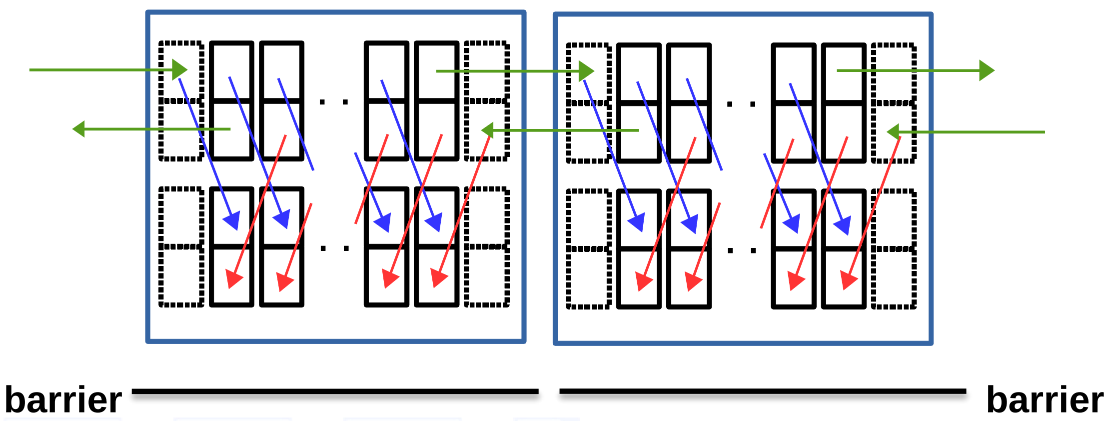
Global
Address Space
Programming Interface
GASPI

Single iteration: details

Left boundary element:

1. Initiate boundary data transfer to remote halo
2. Wait for boundary data transfer to local halo completion
3. Update vector

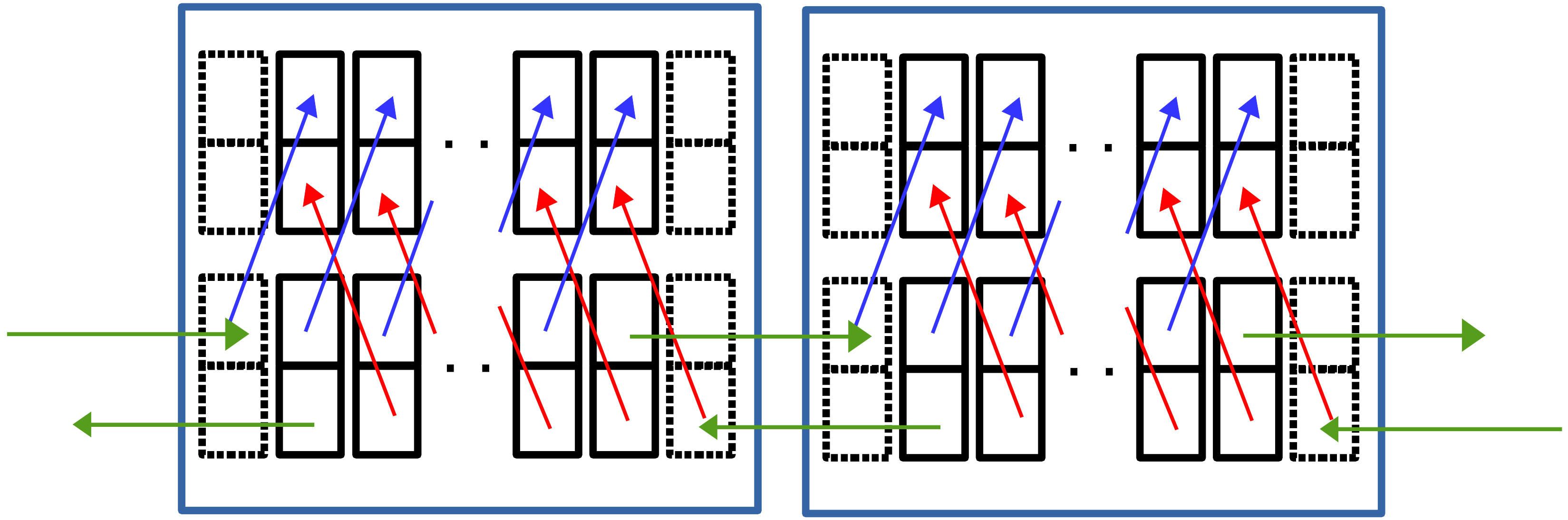
-> Right boundary element handled analogously



In the meanwhile inner elements are done in parallel!

Global
Address Space
Programming Interface
GASPI

Single iteration



barrier barrier

Hands-on

- Implement the overlap of communication and computation
  - use left_right_double_buffer_multiple.c as template

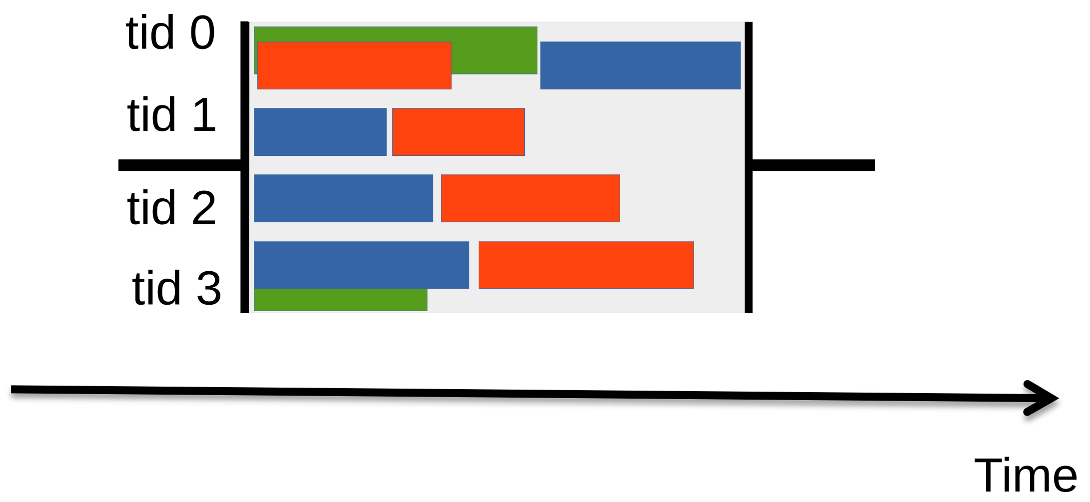
Global
Address Space
Programming Interface
GASPI

The GASPI Ring Exchange

- GASPI – left_right_double_buffer_multiple.c

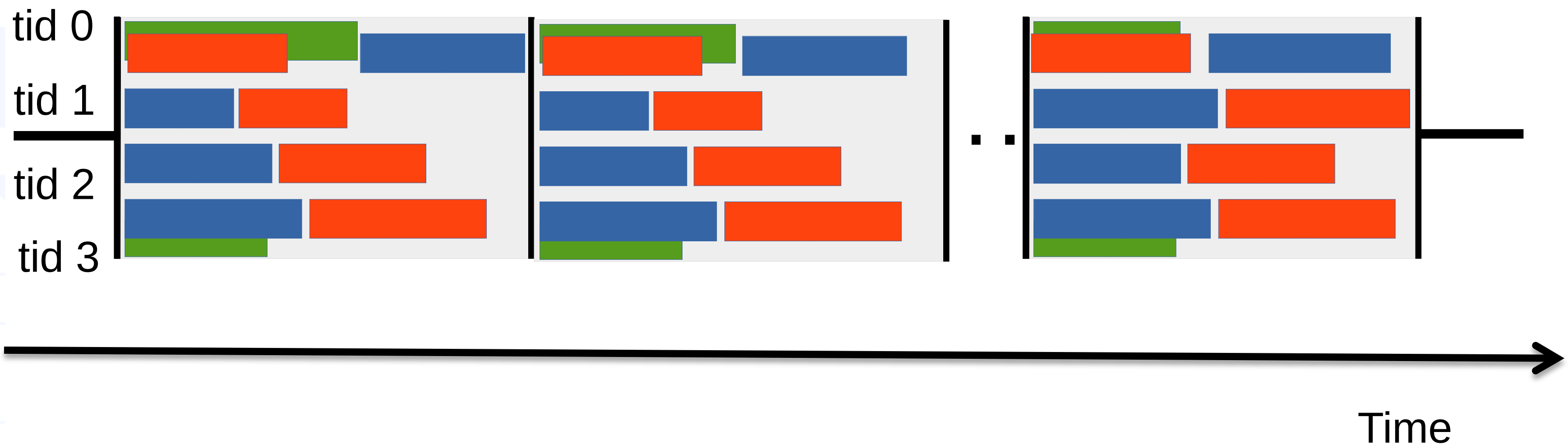
```
if (tid == 0) {
 write_notify_and_cycle
  ( .., LEFT(iProc, nProc),. , right_data_available[buffer_id], 1 + i);
  wait_or_die (segment_id, left_data_available[buffer_id], 1 + i);
  data_compute ( NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
  }
else if (tid < NTHREADS - 1) {
  data_compute ( NTHREADS, array, 1 - buffer_id, buffer_id, slice_id);
 }
else {
 write_notify_and_cycle
  ( .., RIGHT(iProc, nProc),. , left_data_available[buffer_id], 1 + i);
  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;
```

Temporal evolution



Global
Address Space
Programming Interface
GASPI

Temporal evolution



Global
Address Space
Programming Interface
GASPI

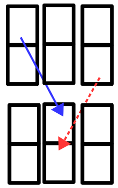
Avoid synchronization point

DATA DEPENDENCY DRIVEN

- What has been achieved?
  - Overlap of communication by computation
  - Communication latency is (partly) hidden
- What has not been achieved?
  - Fully Asynchronous execution
  - Still processwide synchronization after each iteration
    -> process wide aggregation of thread imbalances

Global
Address Space
Programming Interface
GASPI

- Why barrier?
  - Need to know that buffers are ready for next iteration



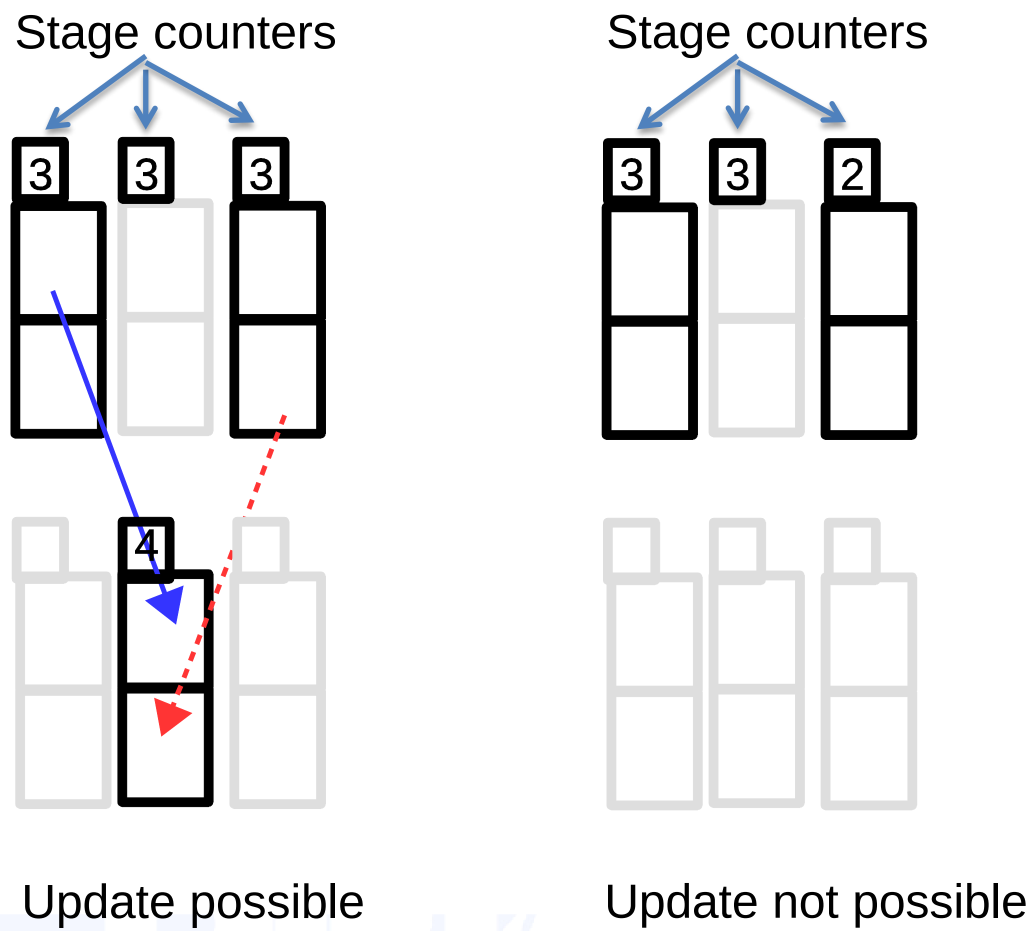
- Barrier provides too much information !!!
  - Only need to know that local neighbours (my dependency) are up to date

Global
Address Space
Programming Interface
GASPI

Reduce synchronicity

- Introduce stage counter for every buffer to account for local states
- check neighbourig stage counters before update
- In case of match: do update
- Increment stage counter after update

-> **Only local dependencies remain**



Global
Address Space
Programming Interface
GASPI

- Avoid static assignment thread / subdomain
  - Instead: „Task“ for each subdomain
    - Compute task for inner subdomain
    - Compute - Initiate data transfer task for boundary subdomains
  - Pre-Condition check before execution
    - Left / right neighbor element do not have a higher iteration counter than me
  - Post-Condition set after execution
    - Increment iteration counter

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;
```

Hands-on

- Implement the data dependency driven algorithm
  - use slice.c as template
  - use left_right_dataflow.c as template

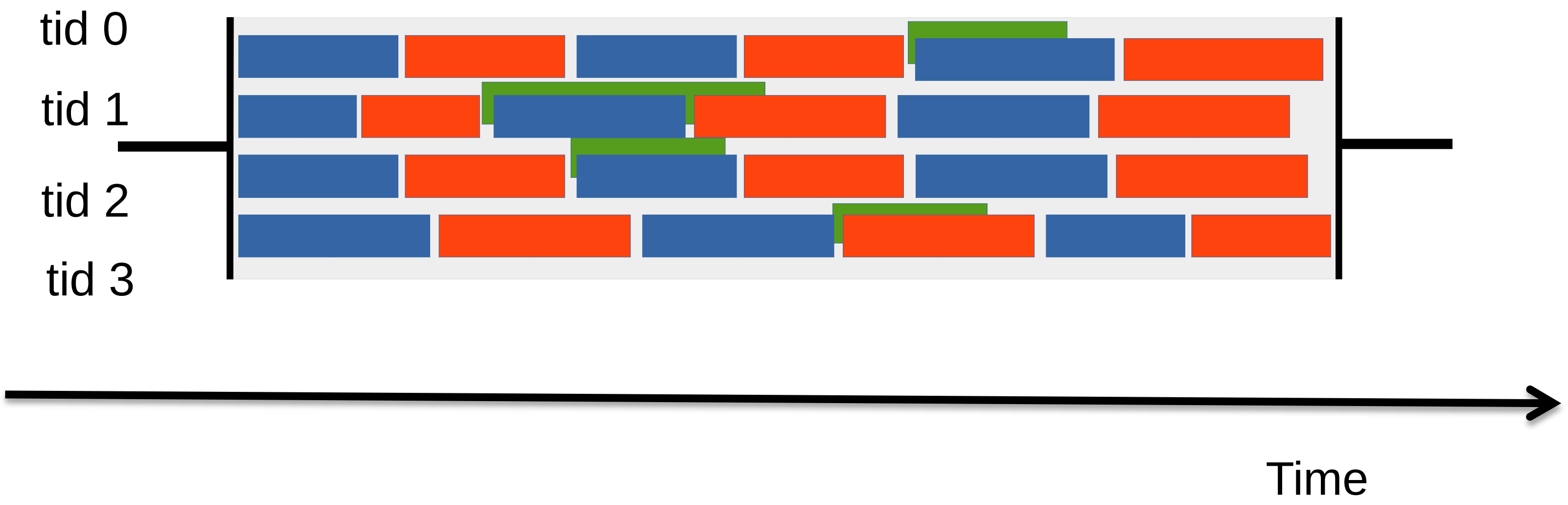
Global
Address Space
Programming Interface
GASPI

The GASPI Ring Exchange

- GASPI – Dataflow - slice.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) {
     write_notify_and_cycle(..);
  } else if (sl->halo_type == RIGHT)
     write_notify_and_cycle(..);
  }
 ++sl->stage;
}
```

Temporal evolution



Global
Address Space
Programming Interface
GASPI

Task (Graph) Models

Bottom up: Complement local task dependencies
with remote data dependencies.

Task (Graph) Models

Targets

- Node local execution on (heterogeneous) manycore architectures.
- Scalability issues in Fork-Join models
- Vertically fragmented memory, separation of access and execution, handling of data marshalling, tiling, etc.
- Inherent node local load imbalance

GASPI

Targets:

- Latency issues, overlap of communication and computation.
- Asynchronous fine-grain dataflow model
- Fault tolerance, system noise, jitter.

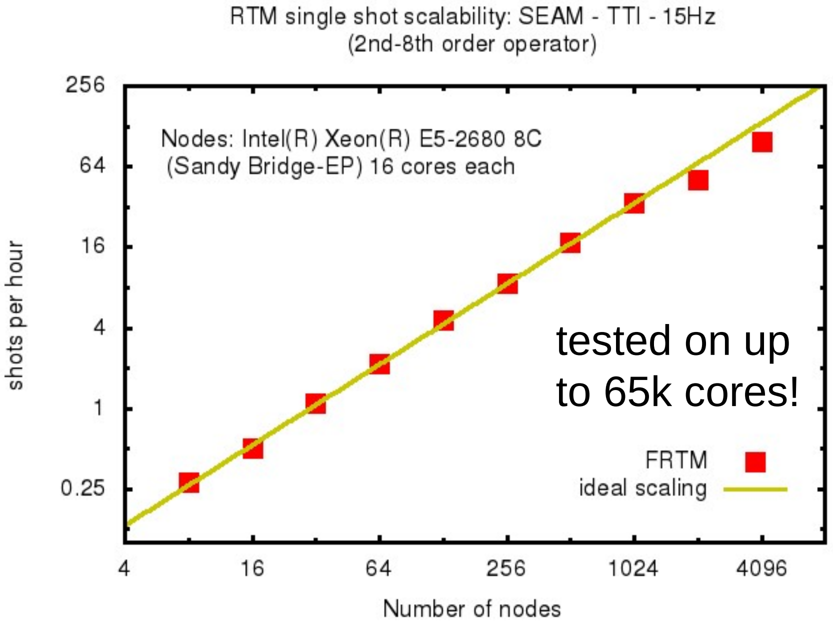
Top Down: Reformulate towards asynchronous dataflow model.
Overlap communication and computation.

Global
Address Space
Programming Interface
GASPI

GASPI at a Glance

Features:

- Global partitioned address space
- Asynchronous, one-sided communication
- Threadsave, every thread can communicate
- Supports fault tolerance
- Open Source
- Standardized API (GASPI)



Infiniband, Cray, Ethernet, GPUs, Intel Xeon Phi, Open Source (GPL) , standardized API

Global
Address Space
Programming Interface
GASPI

Questions?

Thank you for your attention

www.gaspi.de

www.gpi-site.com