Best Practice Guide for the migration of MPI legacy code towards the GASPI-SHAN data-flow model

Version 1.2, 8th Nov 2018

Table of Contents

Table of Figures

1 Introduction

MPI [1] has been used, since its appearance in 1994, as one of the most widespread programming models for distributed memory environments ("Best Practice Guide to Hybrid MPI + OpenMP Programming" [2]) . The API has been evolved through the years in order to include more functionality and adapt himself to take into account new hardware architectures. The standard defines a set of library routines that allow writing portable message-passing programs that usually follow the Single Process Multipe Data (SPMD) execution model, but also supports the Multiple Program Multiple Data execution model. (MPMD)

MPI programs execute multiple processes, each one executing their own code and using the MPI communication primitives in order to explicitly communicate data between nodes and synchronize sections of code running in parallel. Each process executes in its own memory address space. While the MPI standard offers a rich set of features, alternative models have evolved in order to provide improved scalabality or better support for modern hardware architectures. One of these models is the Global Adress Space Programming API (GASPI).

The GASPI standard ("Best Practice Guide for writing GASPI – MPI Interoperable Programs" [2]) promotes the use of one-sided notified communication, where the initiator, has all the relevant information for performing the data movement. GASPI enables the processes to put or get data from remote memory, without engaging the corresponding remote process, or having a synchronization point for every communication request.

GASPI provides weak synchronization primitives which update a notification on the remote side. The corresponding notification semantics is complemented with routines that wait for the updating of a single or a set of notifications. Even though GASPI has been designed to provide interoperability with MPI (in order to allow for an incremental porting of applications), GASPI in general assumes a hybrid application, where threads (or tasks) are responsible for intra-node computation and GASPI then is leveraged for inter-node communication. In consequence today there is no shared memory implementation for GASPI. A migration of 'flat' MPI legacy code (where MPI is being used in shared memory) towards GASPI hence hitherto was rarely successful. In order to overcome these shortcoming the Intertwine project has developed a GASPI shared memory extension (GASPI-SHAred Notifications, GASPI-SHAN). GASPI-SHAN extends the notified communication of GASPI into shared memory windows.

1.1 Purpose

The purpose of this document is to establish a set of advice and guidelines to be used for the migration of 'flat' MPI legacy code towards GASPI-SHAN. The document will serve as a guide for application developers that are considering an improved scalability for their applications via notified intra- and inter-node communication.

1.2 Glossary of Acronyms

GASPI	Global Address Space Programming Interface
SHAN	Shared Notifications
API	Application Programming Interface
SPMD	Single Program Multiple Data
MPMD	Multiple Program Multiple Data
PGAS	Partitioned Global Address Space
SMP	Symmetric Multi-Processor
SOTA	State of the Art

2 Interoperability

2.1 Interoperability between GASPI and MPI, SOTA

While porting from a flat MPI model to the hybrid model of MPI+GASPI is supported in GASPI, hitherto only moderate speedup for the entire application were observed.

When examining these results in more details, it becomes clear that today there are two major shortcomings for a potential migration form a flat MPI model to the hybrid MPI+GASPI model:

The first obstacle is that GASPI has never been designed for a flat process model. Rather the GASPI programming model so far assumes that the application is implemented in some multithreaded model, either task based, with loop level parallelism or otherwise (e.g. with a thread based (sub)-domain decomposition). Communication in GASPI hence usually always is implemented as inter-node communication, rather than intra-node communication.

The second obstacle is that GASPI does not directly support communication data types. The main reason for this is that (for multi-threaded inter-node communication) the concept of data types is rarely efficient: An application which exposes its data layout to the communication layer expects that the required data conversions (from send data type to the linear communication buffer and from a linear communication buffer to the receive data type) have to be either handled by the network adapter or by a (usually single threaded) communication library. Both methods will create a serial bottleneck in the critical communication path as packing and unpacking has to be done by either a single thread or the network adapter.

With the exception of a write_list (for large chunks of data) GASPI hence relies on parallel multithreaded packing and unpacking of data (to and from a linear communication buffer) being handled by the application. **Figure 1** highlights this current state of the art.

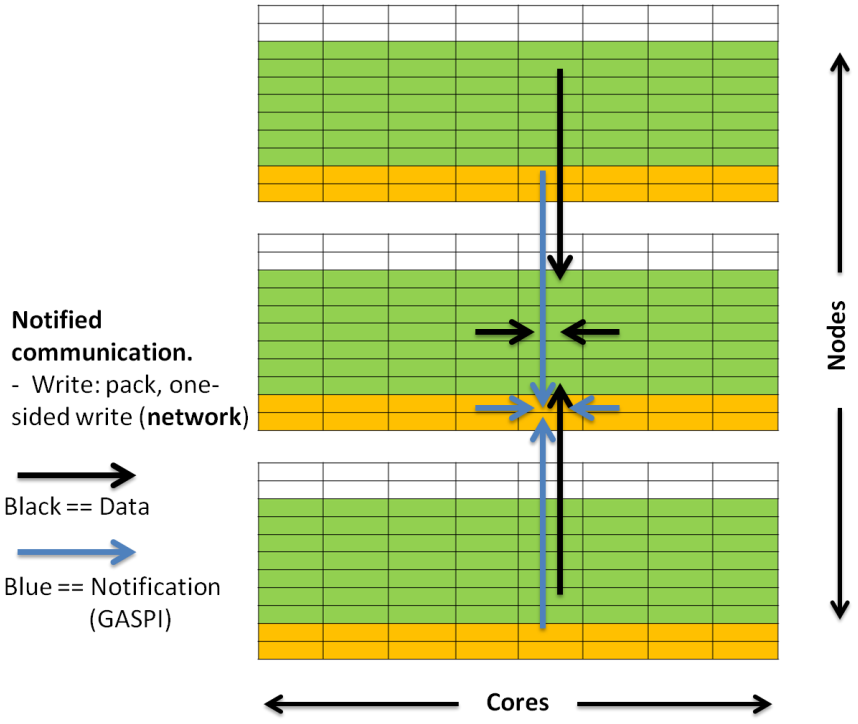

Figure 1: Current State of the Art in hybrid MPI-GASPI communication: All communication is routed through the network. Applications have to pack/unpack to/from a linear communication buffer.

3 SHAN - a shared communication layer for GASPI

From these shortcomings it is clear that GASPI requires an improved interface for GASPI for the migration of legacy flat MPI applications. To that the Intertwine project has developed an API which directly exposes application data and data types in shared memory. While the former concept to some extent has been available in MPI since MPI-3.0 the latter concept does not yet exist. The Interwine project also has extended the notified communication of GASPI to shared memory in the form of SHAred memory Notifications (SHAN). Shared memory notifications are composed or a write fence in combination with an atomic increment. This allows for a very-fine grain mechanism which is suitable for mutual process depedencies in shared memory and nicely complements the remote notified communication mechanism of GASPI (**Figure 2**).

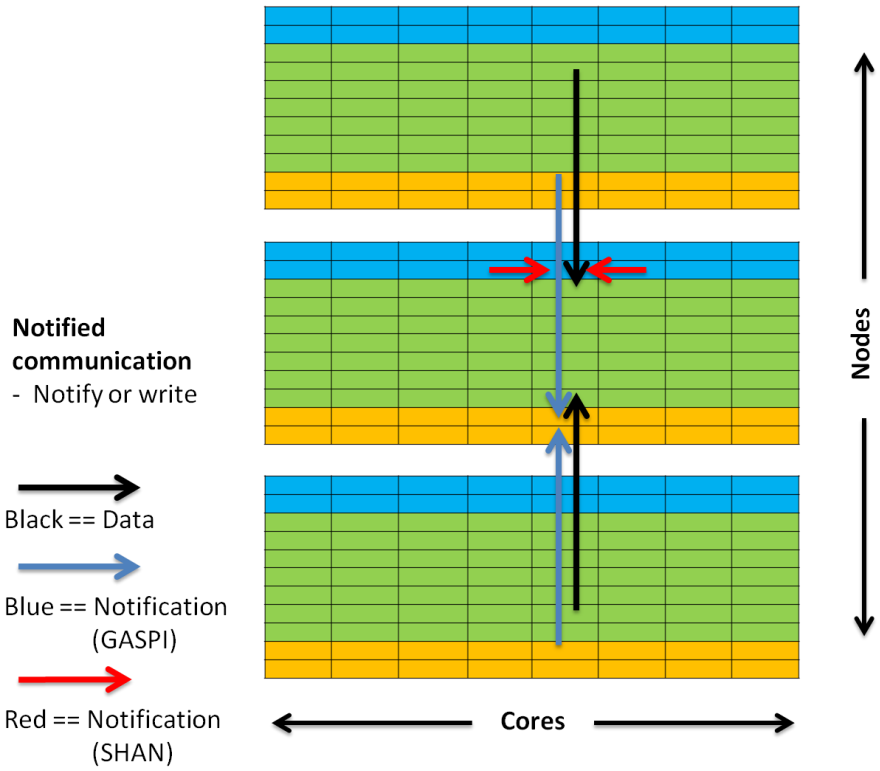

Figure 2: GASPI-SHAN communication: Only remote communication is routed through the network. Local communication is replaced by notifications in shared memory and direct access to the application data.

3.1 The SHAN data type

SHAN exposes the data types of the application in shared memory. While this allows neighboring node-local ranks to directly access data types and actual data, it moreover allows for a dynamic change of data type structures. Sizes of data elements, the number of elements and their offsets can be adjusted on the fly. For remote nodes GASPI SHAN assembles a packet header which contains information about the size of the data elements and their number. (As a remote receiver will access a linear communication buffer, sender offsets are not communicated for remote receivers).

3.2 Communication in SHAN

A GASPI-SHAN application directly reads data from neighboring ranks on the same node, sending of data here is replaced by local shared notifications, which signals that data can be read by neighboring node-local ranks. Sending to remote ranks is being performed by a double buffered one-sided notified GASPI write (write_notify). As all

sending of data is one-sided, the problem of late receivers in 2-sided communication can be entirely avoided.

Receiving of node local data is replaced by first testing the validity of the above 'can read' notification and a subsequent conversion of the remote data types (as these are exposed in shared memory, see above) into a local data type. Also the receive will node-locally trigger a second notification for 'have read'. Receiving of remote data is handled through the testing for completion of remote notified GASPI communication. A received GASPI notification from other ranks guarantees that the associated communication buffer is locally available on the receiving side.

Last not least communication in SHAN requires a confirmation for send. The main reason here is that a process must not rewrite the data other ranks are reading before all neighboring ranks have completed the read process. Testing for a completed send in SHAN is replaced by testing for the 'have read' shared notification from all neighbors. For remote ranks the double buffered communication guarantees a race free operation in bidirectional communication: As a remote communication partner requires a message from a previous communication in order to send, the receive of a message from this communication partner implies that the send of the previous communication buffer is complete and that we hence can rewrite the corresponding send buffer.

3.3 Communication side effects

There are some side effects to the communication in SHAN.

- As the MPI inter-process communication does not expose both data-types and actual data, shared memory communication in MPI requires at least twice the memory bandwidth of the SHAN interface. Intra-node communication in SHAN hence is faster that what we currently see in MPI, even for applications which scale linearly with MPI. As scaling measurements are usually based on single node performance, the intranode communication overhead in MPI does not show up at all in these benchmarks. A corresponding SHAN application will feature the same linear scaling – but with a substantial performance gain.
- Sending of data is one-sided and near instantaneous (except for packing of data for remote targets). While for intra-node communication SHAN merely triggers a process local write barrier with a subsequent increment of an atomic counter, inter-node communication will require packing of the communication buffer with a subsequent one-sided GASPI write_notify.
- An overlap of communication and computation rarely pays of in GASPI-SHAN. The reason here is quite simple: There is no communication, at least not node-internally.

As receiving data is replaced by the node-local transpose of data types and the corresponding "send" process is near instantaneous, communication to remote nodes will be triggered very fast in a communication strategy which triggers sending of data after the required computation is complete. Sending and receiving to/from remote nodes then can usually be overlapped with the node-local data transpose.

- As sending of data is near instantaneous, the "receive" operation (transpose of data types) also can start very fast.
- There is an important caveat here: While the SHAN communication library implements both testing and waiting for receives from selected neighbors (or for all of them), a blocking call for waiting for all possible receives (which is also available as a convenience call) would never complete, unless all required sends have been triggered before entering this call. As this is the same situation in MPI however, we expect that application developers are aware of this issue.

4 The SHAN API

While the SHAN API has of order 35 function calls, in the following we will only describe the most important functions.

4.1 Persistent communication resources

Two principle SHAN types are available, one for data and one for types (SHAN DATA, SHAN TYPE). The MPI_COMM_SHM communicator determines the scope of the shared memory domain. Typically this shared memory communicator is returned from a call to MPI_Comm_split_type.

4.1.1 shan_alloc_shared

The shan_alloc_shared function allocates page aligned memory in shared memory.

```
/** Local allocation of shared memory of size dataSz
 *
 * Note: Memory will be page-aligned.
 *
 * @param segment      - segment handle
 * @param shan_id      - (unique) segment id
 * @param shan_type    - type of allocated memory
 * @param dataSz       - required memory size per rank in byte
 * @param MPI_COMM_SHM - shared mem communicator
 *
 * @return SHAN_SUCCESS in case of success, SHAN_ERROR in case of
error.
 */
int shan_alloc_shared(shan_segment_t *const segment
                      , const int shan_id
                      , const int shan_type
                      , const long dataSz
                      , const MPI_Comm MPI_COMM_SHM
        );
```

4.1.2 shan_get_shared_ptr

the function shan_get_shared_ptr returns the pointer for a node local rank. In most scenarios applications will have to replace existing memory allocation for data they want to communicate with these two function calls.

```
/** Gets shared mem pointer for node local ranks
 *
 * @param segment      - segment handle
 * @param rank         - node local rank id
 * @param shm_ptr      - required memory size per rank in byte
 *
 * @return SHAN_SUCCESS in case of success, SHAN_ERROR in case of
error.
 */
int shan_get_shared_ptr(shan_segment_t * const segment
                        , const int rank
                        , void **shm_ptr
    );
```

4.1.3 shan_free_shared

The shan_free_shared function frees the allocated shared data segment.

```
/** Free shared memory.
 *
 * @param segment      - segment handle
 *
 * @return SHAN_SUCCESS in case of success, SHAN_ERROR in case of
error.
 */
int shan_free_shared(shan_segment_t * const segment);
```

4.1.4 shan_comm_init_comm

The shan_comm_init_comm initializes a given neighborhood. Neighborhoods consist of a set of types and a set of neighbors. If the amount of allocated resources is not identical across the entire neighborhood SHAN will allocate these resources with the maximum value across all participating ranks. Resources have to given per type.

Allocated resources are completely independent. A communication in different types will happen in parallel and completely independent from each other. SHAN types are independent persistent communication channels with a persistent set of neighbors, but potentially varying data types. The number of used data types here can be quite high as resources are only allocated for the given neighborhood.

```
/** Initialize persistant communication for shared mem and GASPI.
 *  requires bidirectional communication for synchronization in
 *  one-sided communication.
 *
 *  A zero length messages will work, no message at all will fail.
 *
 *  - allocates shared and private mem for communication
 *  - figures out local and remote comm partners.
 *  - negotiates remote number of neighbors and comm index
 *
 * @param neighborhood_id - general neighborhood handle
 * @param neighbor_hood_id - neighborhood id
 * @param neighbors     - comm partners (neighbors)
 * @param num_neighbors - num comm partners (neighbors)
 * @param maxSendSz     - max send size for every type.
 * @param maxRecvSz     - max recv size for every type
 * @param max_nelem_send - max number of send elements per type
 * @param max_nelem_recv - max number of recv elements per type
 * @param num_type      - number of types
 * @param MPI_COMM_SHM - MPI shared mem communicator
 * @param MPI_COMM_ALL - embedding of shared communicator
 * @return SHAN_SUCCESS in case of success, SHAN_ERROR in case of
error.
 */
int shan_comm_init_comm(shan_neighborhood_t *const neighborhood_id
                        , int neighbor_hood_id
                        , int *neighbors
                        , int num_neighbors
                        , long *maxSendSz
                        , long *maxRecvSz
                        , int *max_nelem_send
                        , int *max_nelem_recv
                        , int num_type
                        , MPI_Comm MPI_COMM_SHM
                        , MPI_Comm MPI_COMM_ALL
    );
```

4.1.5 shan_comm_free_comm

The function shan_comm_free_comm frees the allocated neighborhood, (all allocations for data types and buffers for remote communication)

```
/** Free communication ressources
 *
 * @param neighborhood_id - general neighborhood handle
 *
 * @return SHAN_SUCCESS in case of success, SHAN_ERROR in case of
error.
 */
int shan_comm_free_comm(shan_neighborhood_t *const neighborhood_id);
```

4.1.6 shan_comm_type_offset

The function shan_comm_type_offset returns, the pointes for offsets, data sizes and the number of elements for all neighbors of the neighborhood. This is type information which then will be visible across the node. Typically this is where application developers have to insert the specific layout for e.g. ghost cell offsets, number of ghost cell elements and element sizes for ghost cells.

```
/** Gets type data structure for node local ranks
*
 * @param neighborhood_id - general neighborhood handle
 * @param type_id         - used type id
 * @param nelem_send - pointer to number of send elements in shared
mem
 * @param nelem_recv - pointer to number of recv elements in shared
mem
 * @param send_sz         - pointer to send size in shared mem
 * @param recv_sz         - pointer to recv size in shared mem
 * @param send_offset     - pointer to offset of send elements in
shared mem
 * @param recv_offset     - pointer to offset of recv elements in
shared mem
 *
 * @return SHAN_COMM_SUCCESS in case of success, SHAN_COMM_ERROR in
case of error.
 */
int shan_comm_type_offset(shan_neighborhood_t *neighborhood_id
                          , int type_id
                          , int **nelem_send
                          , int **nelem_recv
                          , int **send_sz
                          , int **recv_sz
                          , long **send_offset
                          , long **recv_offset
    );
```

4.2 SHAN communication

4.2.1 shan_comm_notify_or_write

The function shan_comm_notify_or_write replaces existing MPI_send calls (as there is no equivalent for MPI_Recv in the on-sided communication pattern of SHAN, MPI_IRECV calls would be removed entirely).

```
/** Writes data or flags data as readable.
 *
 *  - aggregates send data into linear buffer or
 *  - flags data as readable
 *     - number of elements
 *     - element sizes and
 *     - element offsets
 *
 * @param neighborhood_id - general neighborhood handle
 * @param data_segment   - data segment handle
 * @param type_id        - type index
 * @param idx            - comm index for target rank in neighborhood
 *
 * @return SHAN_COMM_SUCCESS in case of success, SHAN_COMM_ERROR in
case of error.
 */
int
shan_comm_notify_or_write(shan_neighborhood_t *const neighborhood_id
                          , shan_segment_t *data_segment
                          , int type_id
                          , int idx
    );
```

4.2.2 shan_comm_wait4All

The shan_comm_wait4All is a convenience call which tests for complete local or remote notifications, and converts remote data types into local types. It also waits until all local neighbors have read the data.

In passing we note that the wait4All function is a convenience call for both “wait for receive (wait4AllRecv)” and “wait for send (wait4AllSend)”, which in turn are convenience functions for testing for individual send and receive requests. The latter are tests for completed notifications. These tests are available for local and remote and exposed to application developers with the SHAN interface. Testing for completed notifications hence can be very grain if the application developers require this functionality.

```
/** Waits for entire neighborhood
 *
 *  - waits for either shared memory notifications
 *    or remote GASPI notifications
 *  - directly converts send type into recv type in shared memory
 *  - unpacks pipelined remote communictation
 *    into the current receive type.
 *  - waits for all receive requests.
 *  - waits for all send requests
 *
 * @param neighborhood_id - general neighborhood handle
 * @param data_segment    - data segment handle
 * @param type_id         - type index
 *
 * @return SHAN_COMM_SUCCESS in case of success, SHAN_COMM_ERROR in
case of error.
 */
 int shan_comm_wait4All(shan_neighborhood_t *const neighborhood_id
                        , shan_segment_t *data_segment
                        , int type_id
    );
```

5 SHAN - Best practice

5.1 Initialization and topology

As GASPI has not been designed for a communication pattern which is multi-process per node rather than multithreaded per node, the SHAN communication library is minimizing the corresponding resource impact and overhead. GASPI here has to be set up with a minimally connected topology.

```
MPI_Init(&argc, &argv);
..
gaspi_config_t config;
SUCCESS_OR_DIE (gaspi_config_get(&config));
config.build_infrastructure = GASPI_TOPOLOGY_NONE;
SUCCESS_OR_DIE (gaspi_config_set(config));
SUCCESS_OR_DIE (gaspi_proc_init (GASPI_BLOCK));
```

Required connections for communication neighborhoods are set up explicitly only for the involved communication partners. The following code shows a corresponding possible function call.

```
MPI_Comm_split_type (MPI_COMM_WORLD
                     , MPI_COMM_TYPE_SHARED
                     , 0
                     , MPI_INFO_NULL
                     , &MPI_COMM_SHM
                     );
```

The SHAN communication library uses an MPI communicator to establish a group of ranks which communicate via shared memory rather than via the network adapter.

5.2 The SHAN type system

In SHAN, the number of types determines the number of independent persistent communication channel. The correspondingly required memory scales with the number of neighbors (per rank). Depending on the number of neighbors resource consumption will be moderate. On the other hand much performance can be gained by enabling independent communication channels. Also – due to the one-sided notified communication - there is very little overhead in both SHAN and GASPI in testing for completion in a multitude of communication channels. There is neither tag-matching overhead nor late receivers. Liberal use of independent communication channels is strongly encouraged both by GASPI and SHAN, whenever communication potentially can progress independently.

```
int const num_type   = nby - 2;
  for (int i = 0; i < num_type; ++i)
  {
      maxSendSz[i] = BSY * sizeof(double);
      maxRecvSz[i] = BSY * sizeof(double);
      max_nelem_send[i] = 1;
      max_nelem_recv[i] = 1;
  }
  int const neighbor_hood_id = 0;
  shan_comm_init_comm(&conf.neighborhood_id
                      , neighbor_hood_id
                      , neighbors
                      , num_neighbors
                      , maxSendSz
                      , maxRecvSz
                      , max_nelem_send
                      , max_nelem_recv
                      , num_type
                      , MPI_COMM_SHM
                      , MPI_COMM_WORLD
      );
```

SHAN assumes that offsets for different neighbors are separated by max_nelem_send and max_nelem_recv neighbors respectively. An initialization for GASPI types hence might assume the following form

```
shan_comm_type_offset(&(cd->neighborhood_id)
                          , 0
                          , &nelem_send
                          , &nelem_recv
                          , &send_sz
                          , &recv_sz
                          , &send_offset
                          , &recv_offset
        );

    for(i = 0; i < cd->ncommdomains; i++)
    {
        int k = cd->commpartner[i];
        nelem_send[i] = cd->sendcount[k];
        nelem_recv[i]  = cd->recvcount[k];
        send_sz[i]         =  max_elem_sz;
        recv_sz[i]         =  max_elem_sz;

        for(j = 0; j < cd->sendcount[k]; j++)
        {
            int pnt = cd->sendindex[k][j];
            ptrdiff_t diff =  &(grad[pnt][0][0])
                            - &(grad[0][0][0]);
            send_offset[i* max_nelem_send  + j] = diff;
        }
        for(j = 0; j < cd->recvcount[k]; j++)
        {
            int pnt = cd->recvindex[k][j];
            ptrdiff_t diff =  &(grad[pnt][0][0])
                            - &(grad[0][0][0]);
            recv_offset[i* max_nelem_recv + j] = diff;
        }
     }
```

The SHAN communication library uses offsets rather than pointers for its data types. As offsets are sometimes not easy to calculate (and even might depend on compilers) it sometimes is useful to calculate the required offsets directly from the corresponding pointer values.

5.3 Freeing resources

Last not least shutting down SHAN and GASPI require corresponding function calls in order to to free the allocate resources.

```
   int res = shan_free_shared(&(sd->dataSegment));
   ASSERT(res == SHAN_SUCCESS);

   int res = shan_comm_free_comm(&(cd->neighborhood_id));
   ASSERT(res == SHAN_SUCCESS);

   SUCCESS_OR_DIE (gaspi_proc_term (GASPI_BLOCK));

   MPI_Finalize ( );
```

5.4 Fortran

Fortran application developers should use the wrapper functions provided by the SHAN communication library. For example for allocation the SHAN wrapper interface returns a C pointer (ISO C BINDING) to the allocated memory. A segment id here can be used instead of the corresponding c structure.

```
       SEGMENT_ID = 0
       SIZE = NUM_VARS*(NX+2)*(NY+2)*(NZ+2)*C_SIZEOF(SZ_REAL)
       CALL F_SHAN_ALLOC_SHARED(SEGMENT_ID, SIZE, CPTR)

       SHAPE = (/ NX+2,NY+2,NZ+2,NUM_VARS /)
       CALL C_F_POINTER(CPTR, GRID, SHAPE)
       GRID(0:, 0:, 0:, 1:) => GRID
```

The SHAN communication library internally uses C convention, where arrays start with 0. For existing Fortran legacy MPI code calls to the SHAN communication library hence usually require a corresponding decrement of indices.

```
 IF ( NEIGHBORS(FRONT) /= -1 ) THEN
       ! Pack and send
       C_IDX = IDX_NEIGHBORS(FRONT) - 1
       CALL F_SHAN_COMM_NOTIFY_OR_WRITE(NEIGHBOR_HOOD_ID,
SEGMENT_ID, &
            TYPE_ID, C_IDX)
  END IF
```

6 References

[1] MPI Documents. Accessed on March 21st, 2017. Available at: https://www.mpi-forum.org/docs/

[2] Best Practice Guide to Hybrid MPI + OpenMP Programming. Milestone of INTERTWinE project.

[3] Best Practice Guide for writing GASPI – MPI Interoperable Programs. Milestone of INTERTWinE project.

