NAAICE Middleware API Documentation

Florian Mikolajczak, Dylan Everingham, Hannes Signer
NAAICE AP2
University of Potsdam, Zuse-Institute Berlin

February 6, 2025

1 Introduction

This document describes the API for the NAAICE Middleware. In NAAICE, network-attached accelerators (NAA) will be integrated into HPC centers using RoCEv2, allowing for IP-based remote direct memory accesses (RDMA). In order for this to work, a middleware/communication library has to be designed. The communication between HPC nodes and NAAs will be implemented as remote asynchronous RPC calls. As low-level libraries `libibvers` and `librdmacm` will be used. A communication library for RDMA communication with NAAs has already been produced. This library includes an RPC-style data transfer, where the end of the data transfer automatically synchronizes both communication partners, making a standalone synchronization protocol unnecessary. After the data transfer is finished, the actual computation starts.
Still, we propose a middleware library to achieve the following goals:

1. **Easy integration into HPC applications**
   In comparison to the already existing communication library, most communication details should be transparent to the user. They can and should be abstracted away from the user, since an application developer should not deal with communication specifics, but rather only specify which computation should be offloaded to an NAA. Communication details include a communication context containing `ibverbs`-specific structures such as Infiniband queue pairs or details of memory region handling.

2. **Fast adaption by the HPC community**
   The middleware library should be easily understood and used by the HPC community. As such, the popular and well known Message Passing Interface (MPI) standard will be taken as an inspiration to formulate a NAA middleware, which will reproduce or reuse functionalities and structures of the MPI standards. We assume that a middleware with analogies to MPI can be more easily adopted by the HPC community.

3. **Ability for communication-computation-overlap (CCO)**
   Instead of serial communication and computation, the aim of the middleware is to allow both to happen at the same time. For this, non-blocking communication calls are necessary. Non-blocking communication functions return before the actual communication i.e. data-transfer is done. The host node can then continue with some other computation while the NIC continues with the data transfer.

2 Reasons for a New Middleware

At the beginning of the middleware development, existing middleware library or standards used within the HPC community were analyzed to ascertain, whether a fully new middleware library was necessary. Two popular standards within the HPC community include MPI and Global Address Space Programming Interface (GASPI). Both allow for two- and one-sided communication. The GASPI implementation GPI-2.0 also specifies RDMA calls [1]. Implementing RDMA for one-sided communications calls is part of the MPI standard. However, popular implementations of MPI such as OpenMPI employ RDMA for one-sided communication calls [1].

2.1 MPI

MPI is a standard for message passing in distributed HPC applications [2]. As such it supports single program multiple data (SPMD) parallelism. An MPI program usually consists of multiple processes which are part of a so-called communicator. The communicator structure is used for addressing between the different processes. A process is identified by its communicator and rank within the communicator. The MPI standard defines a multitude of functions, including ones for setting up/finishing an MPI program and point-to-point or collective operations.

2.2 GASPI

GASPI is another API for distributed HPC applications [1]. As such it implores a Partitioned Global Address Space (PGAS). PGAS is a global memory address space partitioned between the involved local processes. GASPI supports SPMD parallelism like MPI. The communication structure in GASPI is called a group. Within such a group, each process is again given a rank. The group of process and its rank are again used for addressing.

2.3 Conclusion

Both MPI and GASPI revolve around a group of processes that communicate with each other via messages. In MPI this structure is called a communicator, while in GASPI it is known as a group. Usually, in both standards, the processes in a group or communicator do the same calculations with different data and exchange data in between computation segments. However, the NAA will generally not be taking part in a group of processes as an equal member executing the same code as other processes. Rather, the NAA should be used to offload a specific computation task.
Therefore, integrating the NAA as a process in an MPI communicator (and implementing the one-sided communication part of the MPI standard), or analogously doing the same for GASPI, introduces problems with existing HPC applications. These existing applications would need to be changed to treat the NAA-process differently than any other process. To prevent this from happening, we propose a new middleware API just for interacting with the NAA; to connect, exchange data and trigger computation as an asynchronous RPC. However, the functions of our middleware are defined with MPI in mind such that users familiar with MPI will experience an ease of use with our proposed middleware.

[1] OpenMPI 5.0 Documentation

3 NAAICE Middleware API

All routines return an integer of type `naa_err`. Successful routines return 0 on success or a positive integer on failure, indicating the failure reason.

1. `naa_create`
   - Finding IP address and socket ID for an NAA matching required function code. Prepare connection, register and exchange memory region information between HPC node and NAA
   - IP address and socket ID are already known to HPC node. Info is retrieved from resource management system (Slurm) at creation/deployment of slurm job. User knows function code for method/calculation to outsource to NAA. Connection to NAA is done by connection establishment protocol from the Infiniband standard. During connection preparation, the HPC nodes allocates buffers for memory regions and resolves route to NAA. After connection establishment, memory region information is exchanged between HPC node and NAA. The protocol for this was designed in NAAICE AP1 (see documentation for AP1).
   - `C` Binding:

     ```
     int naa_create(uint function_code, naa_param_t *input_params,
     size_t input_params_amount, naa_param_t *output_params,
     size_t output_params_amount, naa_handle *handle);
     ```

     A function code, a list of input and output parameters with their corresponding number of elements and a naa handle must be passed by the user. The function code needs to be defined in advance by the application developers and encodes the functionality to be executed on the receiver side. The list for the input and output parameters is of type `naa_paramt_t` and must always contain the address and the number of bytes to be transferred to or from the NAA for the given parameter (see also Listing 4 for an example). The `singlesend` option, which is disabled by default, can be set as an additional boolean parameter. With this option enabled, the corresponding memory region will only be transferred once during the first RPC call, e.g. to transfer initialization data. This method will register the addresses of the parameters with `ibverbs` as memory regions, hiding the memory region semantic from the user. All memory regions, for both input and output parameters are announced to the NAA during `naa_create()`. Therefore, memory regions can not be changed from input to output between iterations. Currently, no example has been found where this is necessary. The handle object was previously returned by the library and includes information on how to connect to the right NAA. The resource management system will provide information on the IP of the NAA and socket ID of the NAA.

2. `naa_invoke`
   - Goal: Write Data (in 1 or more RDMA operations) to NAA and trigger RPC-like behavior.
   - Application programmer and NAA programmer have to agree on the size and meaning of each memory region beforehand. For now, the syntax and semantics of the transferred data is not explicitly stated.

- Data transfer is done with `RDMA_WITH_IMM`. If the transfer requires $n|n > 1$ operations, $n - 1$ `RDMA_WRITE` operations are done. The last writing operation is `RDMA_WITH_IMM`, where the immediate data value is the function code. `RDMA_WITH_IMM` signals the end of the data transfer to the NAA and initiates calculations on the NAA (RPC start).
- `C` Binding:

  ```
  int naa_invoke(naa_handle *handle);
  ```

  Only the `naa_handle` is passed. Input and output data are fixed during `naa_create()`.

3. `naa_test`
   - Check if result data has been written to HPC node.
   - Much like `MPI_TEST`, the `naa_test` call is non-blocking and polls the completion queue of the queue pair associated with the data transfer.
   - `C` Binding:

     ```
     int naa_test(naa_handle *handle, bool *flag, naa_status *status)
     ```

     A call to `naa_test` returns `flag = true` if the operation identified by `handle` is complete. In such a case, the `status` object is set to contain information on the completed operation. The call returns `flag = false` if the operation is not complete. In this case, the value of the `status` object is undefined.

4. `naa_wait`
   - Check if result data has been written to HPC node.
   - Much like `MPI_WAIT`, the `naa_wait` call is blocking and polls the completion queue of the queue pair associated with the data transfer. `naa_wait` returns, when data has been written back to the HPC node.
   - `C` Binding:

     ```
     int naa_wait(naa_handle *handle, naa_status *status)
     ```

     The call returns, in `status`, information on the completed operation.

5. `naa_finalize`
   - Disconnect in an orderly fashion from the NAA and clean up allocated resources.
   - In `naa_finalize`, the connection between HPC node and NAA is cleaned up as standardized in the Infiniband specification. In addition, resources that were allocated for and during the data transfer are cleaned up. `naa_finalize` is not called after each iteration of a given simulation/job, but only once, when the program terminates and the NAA is no longer needed.
   - `C` Binding:

     ```
     int naa_finalize(naa_handle *handle)
     ```

4 Errorcodes for the RPC

The following errorcodes have been set up.

- 0x01: Socket not available
- 0x02: Kernel timeout
- 0x03-0x0f: reserved
- 0x10 - 0x7f: Application / calculation errors

Pseudocode Example: Vector Addition

```
//All data is gathered and sent to the NAA in one go.
#include <stdlib.h>
#include </sys/socket.h>
#define FNCODE_VEC_ADD 0
typedef struct naa_param_t{
    void *addr;
    size_t size; //number of bytes to be sent
} naa_param_t;

void *a, *b, *c;
a = calloc(64, sizeof(double));
b = calloc(64, sizeof(double));
c = calloc(64, sizeof(double));

set_inputs(a, 64, b, 64) ;

// first input memory region with enabled singlesend option
naa_param_t input_param[2] = {{a, 64 * sizeof(double), true},
                              {b, 64 * sizeof(double), false}};
naa_param_t output_param[1] = {{c, 64 * sizeof(double), false}};
naa_handle handle;

naa_create(FNCODE_VEC_ADD, &input_params, 2, &output_params, 1, &handle) ;

int flag = 0;
naa_status status;
naa_invoke(&handle);

naa_test(&handle,&flag,&status)
while (!flag) {
  do_other_work();
}
process_results(c);
// set inputs with new data
set_inputs(a, 64, b, 64) ;

naa_invoke(&handle);
//wait for RPC to finish
naa_wait (&handle,&status)
process_results(c);

naa_finalize(&handle);
```

References

[1] Daniel Grünewald and Christian Simmendinger. The gaspi api specification and its implementation gpi 2.0. In *7th International Conference on PGAS Programming Models*, volume 243, page 52, 2013.

[2] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.