An API Proposal for Including the NAM into the MPI World

Draft / Version 0.6 / 16-September-2019

Abstract This document presents a draft for a user manual that describes the proposed functions and semantics for accessing Network Attached Memory (NAM) in the DEEP-EST prototype via an extended MPI interface.

Introduction

One distinct feature of the DEEP-EST prototype will be the Network Attached Memory (NAM): Special memory regions that can directly be accessed via Put/Get-operations from every node within the Extoll network. For utilizing such RMA operations for accessing the NAM from within an application, a new version of the libNAM will be available that features a low-level API for doing so. However, for making the programming more convenient and/or familiar, Task 6.1 strives for integrating an interface for accessing the NAM also in the common MPI world. That way, application programmers shall be able to use well-know MPI functions (in particular those of the MPI RMA interface) for accessing NAM regions quite similar to other remote memory regions in a standardized (or at least harmonized) fashion under the single roof of an MPI world. In doing so, we always try to stick to the current MPI standard as close as possible and to avoid the introduction of new API functions wherever possible. For readability, we denote API extensions with the prefix “MPIX” and other semantical additions with the prefix “deep” within this document.

Acquiring NAM Memory

General Semantics

The main issue when mapping the libNAM API onto the MPI RMA interface is the fact that MPI assumes that all target and/or origin memory regions for RMA operations are always associated with an MPI process being the owner of that memory. That means that in an MPI world, remote memory regions are always addressed by means of a process *rank* (plus handle, which is the respective *window* object, plus offset), whereas the libNAM API will most probably merely require an opaque handle for addressing the respective NAM region (plus offset). Therefore, a mapping between remote MPI ranks and the remote NAM memory needs somehow to be realized. According to this proposal, this correlation will be achieved by sticking to the notion of an *ownership* in a sense that definite regions of the NAM memory space are *logically* assigned to particular MPI ranks. However, it has to be emphasised that this is a purely software-based mapping being conducted by the MPI layer. That means that the related MPI window regions (though globally accessible and located within the NAM) have then to be addressed by means of the rank of that process to which the NAM region is assigned.

Semantics Terms

At this point, the semantic terms of memory *allocation*, memory *region* and memory *segment* shall be determined for their usage within this proposal. The reason for this is that, for example, the term "allocation" is commonly used for both: a resource as granted by the job scheduler and a memory region as returned e.g. by malloc. Therefore, we need a stringent nomenclature here:

NAM Memory Allocation
: A certain amount of (most probably contiguous) NAM memory space that has been requested and granted (e.g. through the job scheduler) for the certain MPI session.

NAM Memory Segment
: A certain amount of (probably contiguous) NAM memory space that is part of a NAM allocation. According to this, a NAM allocation can logically be subdivided (e.g. by the MPI layer) into multiple memory segments, which can then, for example, be assigned to MPI RMA windows.

NAM Memory Region
: A certain amount of contiguous NAM memory space that is associated to a certain MPI rank in the context of an MPI RMA window.

Rational For performance and also for management reasons, allocation requests towards the NAM and/or the resource manager should preferably occur rarely – so, for instance, only once at the beginning of an MPI session. In order to provide MPI applications with the ability to handle multiple MPI RMA windows within such an allocation, the MPI library presumably needs to implement a further layer of memory management that allows for a logical acquiring and releasing of NAM segments within the limits of the granted allocation.

Interface Specification

For acquiring[1] memory regions on the NAM, we propose semantic extensions to the well-known MPI_WIN_ALLOCATE function:

MPI_WIN_ALLOCATE(size, disp_unit, info, comm, baseptr, win)

IN	size	size of window in bytes (non-negative integer)
IN	disp_unit	local unit size for displacements, in bytes (positive integer)
IN	info	info argument (handle)
IN	comm	intra-communicator (handle)
OUT	baseptr	initial address of window (choice)
OUT	win	window object returned by the call (handle)

Rational This is a collective call executed by all processes in the group of *comm*, which makes it for the MPI library much easier to treat the set of allocated memory regions as an entity.

Alternative We might also think about the possibility to let each process allocate the NAM memory before and to use MPI_WIN_CREATE to assemble the window object thereof. However, as this approach would even allow for mixed windows (i.e. windows mixed-up of NAM and host-local memory regions), we refrain for this approach—at least in the first instance—as it would make the window handling much more complicated for the MPI layer.

[1] Please note that we intentionally use the term "to acquire" here instead of the ambiguous term "to allocate".

According to the actual MPI standard, the MPI_WIN_ALLOCATE function allocates on each process of the calling group a local memory region and returns a pointer to it as well as a window object that can then be used to perform RMA operations. For acquiring NAM regions instead of local memory, we propose to utilize the info argument for telling the MPI library to do so: When setting the key/value pair *deep_mem_kind=deep_nam*, the MPI-internal memory management will determine a still available (probably contiguous) NAM segment within the NAM allocation of the respective job for becoming the memory space for this new RMA window. In doing so, the segment will naturally be subdivided in terms of the *np* NAM regions (*np* = number of processes in *comm*) that form the RMA window from the application's perspective.

Advice to users Please note that application programmers can easily create also "flat" RMA windows by letting only rank 0 passing a *size* argument greater that zero for the call. In such a case, all RMA operations on this window would then have to be addressed to target rank = 0.

Alternatives We could also think about providing a new MPI function, for example like MPIX_WIN_ALLOCATE_NAM or MPIX_WIN_ALLOCATE_REMOTE instead of utilizing the info argument. Such a new function may then also omit the *baseptr* parameter as this is only usable for host-local memory (and might be set to NULL for MPI_WIN_ALLOCATE in the NAM case).

Example

```
MPI_Info_create(&info);
MPI_Info_set(info, "deep_mem_kind", "deep_nam");
MPI_Win_allocate(sizeof(int) * ELEMENTS_PER_PROC, sizeof(int), info, comm, NULL, &win);
…
MPI_Win_fence(0, win);
for(int i=0; i<comm_size; i++) MPI_Put(&i, 1, MPI_INT, i, 0, 1, MPI_INT, win);
MPI_Win_fence(0, win);
…
```

Releasing and Managing NAM Memory

According to this proposal, NAM regions allocated via MPI_WIN_ALLOCATE will be freed (if not persistent, see below) by the collective call of MPI_WIN_FREE. At this point it has to be noted that acquiring and releasing NAM segments repeatedly are assumed to be valid operations in the scope of the granted NAM allocation. That means that this proposal does not deal with the question of how the NAM is managed as a globally contested resource but rather presumes that the MPI application is allowed to request for a certain amount of NAM space and that—as long as this limit is not exceeded—subsequent NAM requests and releases can randomly be conducted within this granted scope.

Advice to users A sound MPI application should free the window (and thus the allocated NAM regions) before calling MPI_FINALIZE.

Advice to implementers Of course, the proposed semantics requires the above mentioned additional memory management layer within the MPI library for handling multiple NAM memory segments (be it already assigned or available again) within the NAM allocation.

Acquiring Persistent NAM Memory

General Semantics

A central use-case for the NAM in DEEP-EST will be the idea of facilitating workflows between different applications and/or application steps. For doing so, the data once put into NAM memory shall later be re-usable by other MPI applications. Of course, this requires that NAM regions—and hence also their related MPI windows—can somehow be denoted as "persistent" so that their content gets not be wiped when the window is freed.

Interface Specification

According to our proposal, a NAM-based MPI window becomes persistent in this sense if it is allocated with an info object that contains *deep_mem_kind=deep_nam_persistent* as a key/value pair. If the creation of the persistent NAM window was successful, the related NAM regions become addressable as a joint entity by means of a *logical* port name. This *port name* can then in turn be retrieved by querying the info object attached to that window afterwards via the info key *deep_win_port_name*.

Advice to users The port name is not to be confused with an actual TCP port or similar things. It's merely a string-coded handle that serves for the MPI library and its runtime for identifying a joint set of NAM regions.

If an MPI application wants to pass data via such a persistent window to a subsequent MPI application, it merely has to pass this port name somehow to its successor so that this other MPI session can then re-attach to the respective window (see below).

Advice to users The passing of this port name could, for example, be done via standard I/O, via command line arguments, or even via MPI-based *name publishing*—as it is shown here in the example code. As the knowledge about this string allows other MPI sessions to attach and to modify the data within the persistent window, it is the responsibility of the application programmer to ensure that data races are avoided—for example, by locally releasing the window via MPI_WIN_FREE before publishing the port name.

Example

```
MPI_Info_create(&info);
MPI_Info_set(info, "deep_mem_kind", "deep_nam_persistent");
MPI_Win_allocate(sizeof(int) * ELEMENTS_PER_PROC, sizeof(int), info, comm, NULL, &win);
MPI_Info_free(&info);

MPI_Win_get_info(win, &info);
MPI_Info_get(info, "deep_win_port_name", INFO_VALUE_LEN, info_value, &flag);
if(flag) {              strcpy(port_name, info_value);
                        printf("The window's port name: %s\n", port_name);
} else {                printf("No port name found!\n");
                        MPI_Abort(MPI_COMM_WORLD, -1);
}
…
// Work on window…
…
```

```
MPI_Win_free(&win);
if(comm_rank == 0) {
                    sprintf(service_name, "%s:my-peristent-nam-window", argv[0]);
                    MPI_Publish_name(service_name, MPI_INFO_NULL, port_name);
}
```

Freeing and Managing Persistent NAM Regions

MPI window objects are commonly to be freed by MPI_WIN_FREE before MPI_FINALIZE is called—according to our proposal, this also holds for NAM windows that are marked as persistent. However, such windows are then merely freed from the perspective of the MPI application, not from the view of the process manager that may handle several (parallel or subsequent) MPI sessions within a single resource allocation.

Alternatives We could also postulate that for persistent window objects, the MPI_WIN_FREE call is to be omitted. However, the above-mentioned solution seems to be sounder.

Rational According to this, there are different degrees with respect to the lifetime of an MPI window: Common MPI windows just live as long as MPI_WIN_FREE has not been called and the related session is still alive. In contrast to this, persistent NAM windows exist as long as the resources (set of nodes and assigned NAM space) are granted by the resource manager.

Attaching to Persistent NAM Regions

Obviously, there needs to be a way for subsequent MPI sessions to attach to the persistent NAM regions previous MPI sessions have created. According to this proposal, this is to be done via a call of MPI_COMM_CONNECT, which is normally used for establishing communication between distinct MPI sessions:

MPI_COMM_CONNECT(port_name, info, root, comm, newcomm)

IN	port_name	network address (string, used only on root)
IN	info	implementation-dependent information (handle, used only on root)
IN	root	rank in comm of root node (integer)
IN	comm	intracommunicator over which call is collective (handle)
OUT	newcomm	intercommunicator with server as remote group (handle)

When passing a valid port name of a persistent NAM window plus an info argument with the key *deep_win_connect* and the value *true*, this function will return an inter-communicator that then serves for accessing the remote NAM memory regions.

Advice to users The returned inter-communicator is just a *pseudo* communicator that cannot be used for any point-to-point or collective communication, but that rather acts like a handle for RMA operations on a virtual window object embodied by the remote NAM memory.

In doing so, the original segmentation of the NAM window is being retained. That means that the window is still divided (and thus addressable) in terms of the MPI ranks of that process group that created the window before. Therefore, a call to MPI_COMM_REMOTE_SIZE on the returned inter-communicator reveals the former number of processes in that group. For actually creating the local representative for the window in terms of an MPI_WIN datatype, we propose to alienate the MPI_WIN_CREATE_DYNAMIC function with the inter-communicator as the input and the window handle as the output parameter.

Alternative Alternatively, we might want to provide a new function like MPIX_WIN_CONNECT that combines the two calls of MPI_COMM_CONNECT and MPI_WIN_CREATE_DYNAMIC.

Querying Information about the Remote Window

After determining the size of the former progress group via MPI_COMM_REMOTE_SIZE, there is still a demand for getting the information about the remote region sizes as well as the related unit sizes for the displacement. According to our proposal, the MPI library provides the following new window attributes for querying these information: MPIX_WIN_SIZES, MPIX_WIN_DISP_UNITS. In contrast to their regular counterparts (MPI_WIN_SIZE and MPI_WIN_DISP_UNIT), these new attributes will return *arrays* filled with the related information to be indexed by the remote ranks.

Advice to users Please not that the required information can also be passed on different ways to the processes—even by putting them into the persistent window right after the initial creating. Of course, such other ways are most probably much trickier to handle, but if the application wants to go without the usage of any new symbols, then this is still possible.

Alternative We may also want to provide a function like MPIX_WIN_QUERY—similar to the existing MPI_WIN_SHARED_QUERY function—that returns the respective information by taking the remote rank as an input parameter.

Example

```
MPI_Info_create(&win_info);
MPI_Info_set(win_info, "deep_win_connect", "true");
MPI_Comm_connect(port_name, info, 0, MPI_COMM_WORLD, &inter_comm);
MPI_Info_free(&info);

printf("Connection to persistent memory region established!\n");
MPI_Comm_remote_size(inter_comm, &group_size);
printf("Number of former process group that created the NAM window: %d\n", group_size);
MPI_Win_create_dynamic(MPI_INFO_NULL, inter_comm, &win);
…
MPI_Win_get_attr(win, MPIX_WIN_SIZES, &win_size_attr, &flag);
for(int i=0; i<group_size; i++) printf("[%d] segment size: %lld\n", i, win_size_attr[i]);
MPI_Win_get_attr(win, MPIX_WIN_DISP_UNITS, &win_disp_unit_attr, &flag);
for(int i=0; i<group_size; i++) printf("[%d] displacement unit %d\n", i, win_disp_unit_attr[i]);
…
```