Manual: Using the NAM with ParaStation MPI

Version 0.1 / December 2020

Abstract *This document presents a user manual for functions and semantics as implemented in the DEEP-EST project for accessing Network Attached Memory (NAM) via the Message-Passing Interface (MPI) and related extensions, referred to as PSNAM.*

Introduction

One distinct feature of the DEEP-EST prototype is the Network Attached Memory (NAM): Special memory regions that can directly be accessed via Put/Get-operations from any node within the Extoll network. For executing such RMA operations on the NAM, a new version of the libNAM is available to the users that features a corresponding low-level API for this purpose. However, to make this programming more convenient—and in particular to also support *parallel* access to shared NAM data by multiple processes—an MPI extension with corresponding wrapper functionalities to the libNAM API has also been developed in the DEEP-EST project. This extension, which is called PSNAM, is a complementary part of the ParaStation MPI—which is the MPI library of choice in the DEEP-EST project—and is as such also available to users on the DEEP-EST prototype system. In this way, application programmers shall be enabled to use known MPI functions (especially those of the MPI RMA interface) for accessing NAM regions in a standardized (or at least harmonized) way under the familiar roof of an MPI world. In doing so, the PSNAM extensions try to stick to the current MPI standard as close as possible and to avoid the introduction of new API functions wherever possible.

Acquiring NAM Memory

General Semantics

The main issue when mapping the MPI RMA interface onto the libNAM API is the fact that MPI assumes that all target and 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 merely requires 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. In PSNAM, this is 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 PSNAM wrapper 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.

Semantic Terms

At this point, the semantic terms of memory *allocation*, memory *region* and memory *segment* are to be determined for their use 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 stricter nomenclature here:

NAM Memory Allocation

A certain amount of contiguous NAM memory space that has been requested from the NAM Manager (and possibly granted through the job scheduler) for an MPI session.

NAM Memory Segment

A certain amount of contiguous NAM memory space that is part of a NAM allocation. According to this, a NAM allocation can logically be subdivided by the PSNAM wrapper layer into multiple memory segments, which can then again 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.

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, PSNAM implements 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 assigning memory regions on the NAM with MPI RMA windows, a semantic extension to the well-known MPI_WIN_ALLOCATE function via its MPI *info* parameter can be used:

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

IN	size	size of memory region in bytes (non-negative integer, may differ between processes)
IN	disp_unit	local unit size for displacements, in bytes (positive integer)
IN	info	info argument (handle) with psnam info keys and values
IN	comm	intra-communicator (handle)
OUT	baseptr	always NULL in case of PSNAM windows
OUT	win	window object returned by the call (handle)

MPI_WIN_ALLOCATE is a collective call to be executed by all processes in the group of *comm*. This in turn enables the PSNAM wrapper layer to treat the set of allocated memory regions as an entity and logically link the regions to a shared RMA window.

The semantic extension compared to the MPI standard is the evaluation of the following keys within the given MPI info object:

- *psnam_manifestation*
- *psnam_consistency*
- *psnam_structure*

The *psnam_manifestation* key specifies which memory type shall be used for a region. The value for using the NAM is *psnam_manifesation_libnam*—but it should be mentioned that also node-local persistent shared-memory (*psnam_manifestation_pershm*) can here be chosen as another supported manifestation. In fact, each process in *comm* can even select a different manifestation of these two for the composition of the window.

The *psnam_consistency* key specifies whether the memory regions of an RMA window shall be persistent (*psnam_consistency_persistent*) or whether they shall be released during the respective MPI_WIN_FREE call (*psnam_consistency_volatile*). This key must be selected equally among all processes in *comm*.

The *psnam_structure* key specifies the memory layout as formed by the multiple regions of an MPI window. Currently, the following three different memory layouts are supported:

- *psnam_structure_raw_and_flat*
- *psnam_structure_managed_contiguous*
- *psnam_structure_managed_distributed*

The chosen memory layout also decides whether and how the PSNAM layer stores further meta data in the NAM regions to allow a later recreation of the structure while reconnecting to a persistent RMA window by another MPI session. The chosen structure must be the same for all processes in *comm*.

Raw and Flat
The *psnam_structure_raw_and_flat* layout is intended to store raw data (i.e. untyped data) in the NAM without adding meta information. According to this layout, only rank 0 of *comm* is allowed to pass a *size* parameter greater than zero during the MPI_WIN_ALLOCATE call. Hence, only rank 0 allocates one (contiguous) NAM region forming the window and all RMA operations on such a flat window have therefore to be addressed to target rank = 0.

Managed Contiguous
In the *psnam_structure_managed_contiguous* case, also only rank 0 allocates (contiguous) NAM space, but this space is then subdivided according to the *size* parameters as passed by all processes in *comm*. That means that here also processes with *rank > 0* can pass a *size > 0* and hence acquire a rank-addressable (sub-)region within this window. Furthermore, the information about the number of processes and the respective region sizes forming that window is being stored as meta data within the NAM. That way, a subsequent MPI session re-connecting to this window can retrieve this information and hence recreate the former structure of the window.

Managed Distributed
In a *psnam_structure_managed_distributed* window, each process that passes a *size > 0* also allocates NAM memory explicitly and on its own. It then contributes this memory as a NAM region to the RMA window so that the corresponding NAM allocation becomes directly addressable by the respective process rank. The following Figure to illustrates the differences between these three structure layouts.

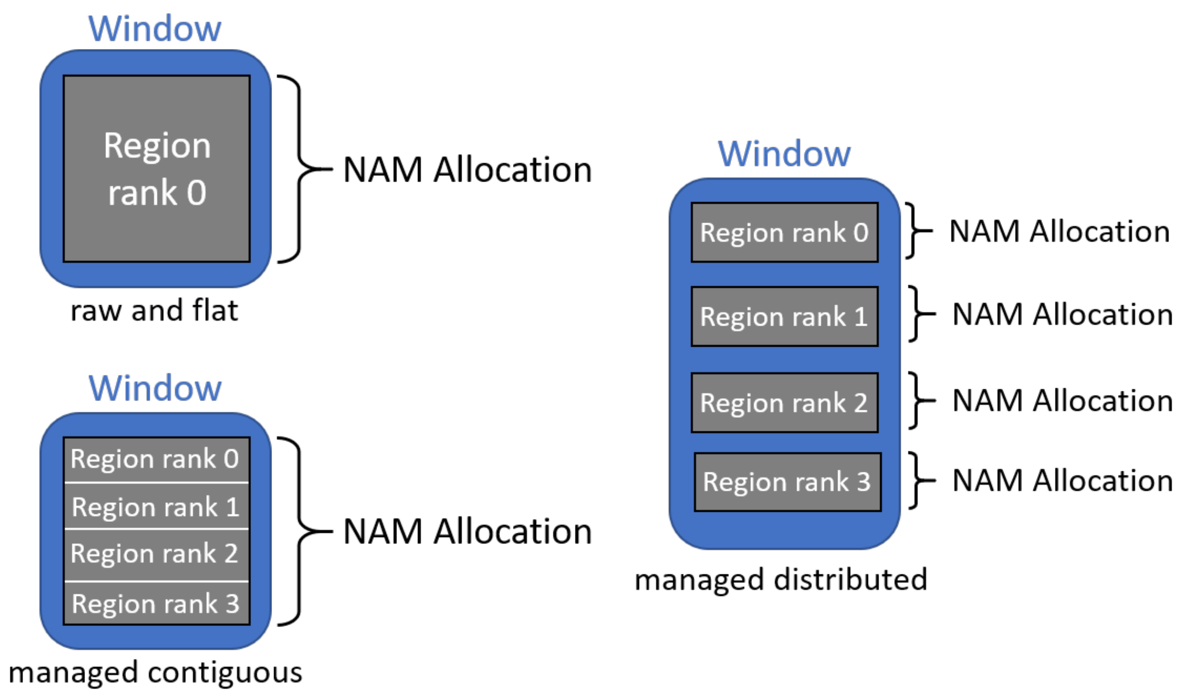

Figure: The different layouts of the three selectable PSNAM window structures

Examples

```
MPI_Info_create(&info);
MPI_Info_set(info, "psnam_manifestation", "psnam_manifestation_libnam");
MPI_Info_set(info, "psnam_consistency", "psnam_consistency_volatile");

// Allocate a "raw_and_flat" window:
MPI_Info_set(win, "psnam_structure", "psnam_structure_raw_and_flat");
MPI_Win_allocate(rank ? 0 : win_size, 1, info, comm, NULL, &win_flat);

// Put some data into the "raw_and_flat" window:
MPI_Win_fence(0, win_flat);
if (rank == 0)
        MPI_Put(data_ptr, win_size, MPI_BYTE, 0 /*=target*/, 0 /*=offset*/, win_size, MPI_BYTE, win_flat);
MPI_Win_fence(0, win_flat);
…

// Allocate a "managed_distributed" window:
MPI_Info_set(win, "psnam_structure", "psnam_structure_ managed_distributed");
MPI_Win_allocate(my_region_size * sizeof(int), sizeof(int) , info, comm, NULL, &win_dist);

// Put some data into the "managed_distributed" window:
MPI_Win_fence(0, win_dist);
MPI_Put(data_ptr, my_region_size, MPI_INT, my_rank, 0 /*=offset*/, my_region_size, MPI_INT, win_dist);
MPI_Win_fence(0, win_dist);
…
```

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 and/or sessions. 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. In fact, this can be achieved by setting the above mentioned *psnam_consistency_persistent* MPI info key when calling MPI_WIN_ALLOCATE.

Window Names

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* name that is system-wide unique. This *window name* can then in turn be retrieved by querying the info object attached to that window afterwards via the info key *psnam_window_name*.

If an MPI application wants to pass data via such a persistent window to a subsequent MPI application, it merely has to pass this window name somehow to its successor so that this other MPI session can then re-attach to the respective window. The passing of this window name could, for example, be done via standard I/O, via command line arguments, or even via MPI-based name publishing. 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 window name.

Example

```
MPI_Info_create(&info);
MPI_Info_set(info, "psnam_consistency", "psnam_consistency_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, "psnam_window_name", INFO_VALUE_LEN, info_value, &flag);
if(flag) {          strcpy(window_name, info_value);
                    printf("The window's name is: %s\n", window_name);
} else {            printf("No psnam window 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-psnam-window", argv[0]);
                    MPI_Publish_name(service_name, MPI_INFO_NULL, window_name);
}
```

Releasing NAM Memory

According to the standard, an MPI RMA window must be freed by the collective call of MPI_WIN_FREE. In case of a PSNAM window, the selection of the *psnam_consistency* MPI info key decided whether the corresponding NAM memory regions are to be freed, too. Since MPI_WIN_FREE has no info parameter, the corresponding selection has either already to be made when calling MPI_WIN_ALLOCATE and/or can also be made/changed later by using MPI_WIN_INFO_SET.

A sound MPI application must free all MPI window objects before calling MPI_FINALIZE—regardless whether the corresponding NAM region should be persistent or not. 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 assigned NAM space is granted by the NAM manager. Upon an MPI_WIN_FREE call, such windows are merely freed from the perspective of the MPI current application, not from the view of the NAM 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. The PSNAM wrapper layer enables this to be done via a call to MPI_COMM_CONNECT, which is normally used for establishing communication between distinct MPI sessions:

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

IN	window_name	globally unique window name (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 name of a persistent NAM window plus an info argument with the key *psnam_window_connect* and the value *true*, this function will return an inter-communicator that then serves for accessing the remote NAM memory regions. However, this 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 structure 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, the MPI_WIN_CREATE_DYNAMIC function can be used with the inter-communicator as the input and the window handle as the output parameter.

Querying Information about a Remote Window

After determining the size of the former progress group via MPI_COMM_REMOTE_SIZE, there might also be a demand for getting the information about the remote region sizes as well as the related unit sizes for displacements. For this purpose, the PSNAM wrapper hooks into the MPI_WIN_SHARED_QUERY function that returns these values according to the passed rank:

MPI_WIN_SHARED_QUERY (win, rank, size, disp_unit, baseptr)

IN	win	window object used for communication (handle)
IN	rank	remote region rank
OUT	size	size of the region at the given rank
OUT	disp_unit	local unit size for displacements at the given rank (in bytes)
OUT	baseptr	always NULL in case of PSNAM windows

Example

```
MPI_Info_create(&win_info);
MPI_Info_set(win_info, "psnam_window_connect", "true");
MPI_Comm_connect(window_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, &remote_group_size);
printf("Number of former process group that created the NAM window: %d\n", remote_group_size);
MPI_Win_create_dynamic(MPI_INFO_NULL, inter_comm, &win);
…
For (int region_rank=0; region_rank < remote_group_size; region_rank++) {
        MPI_Win_shared_query(win, region_rank, &region_size[i], &disp_unit[i], NULL);
}
…
```

Pre-Allocated NAM Memory and Segments

Without further info parameters than described so far, MPI_WIN_ALLOCATE will always try to allocate NAM memory itself and “on-demand”. However, a common use case might be that the required NAM memory needed by an application has already been allocated beforehand via the batch system—and the question is how such pre-allocated memory can be handled on MPI level. In fact, using an existing NAM allocation during an MPI_WIN_ALLOCATE call instead of allocating new space in quite straight forward by applying *psnam_libnam_allocation_id* as a further info key plus the respective NAM allocation ID as the related info value.

Usage of Segments

However, a NAM-based MPI window may possibly still consist of multiple regions, and it should also still be possible to build multiple MPI windows from the space of a single NAM (pre-)allocation. Therefore, a means for subdividing NAM allocations needs to be provided—and that's exactly what *segments* are intended for: A segment is a “meta-manifestation” that maintains a size and offset information for a sub-region within a larger allocation. This offset can either be set explicitly via *psnam_segment_offset* (e.g., for splitting an allocation among multiple processes), or it can be managed dynamically and implicitly by the PSNAM layer (e.g., for using the allocated memory across multiple MPI windows).

Recursive Usage

The concept of segments can also be applied recursively. For doing so, PSNAM windows of the “raw and flat” structure feature the info key *psnam_allocation_id* plus respective value that in turn can be used to pass a reference to an already existing allocation to a subsequent MPI_WIN_ALLOCATE call with *psnam_manifestation_segment* as the region manifestation. That way, existing allocations can be divided into segments—which could then even further sub-divided into sub-sections, and so forth.

Example

```
MPI_Info_create(&info_set);
MPI_Info_set(info_set, "psnam_manifestation", "psnam_manifestation_libnam");
MPI_Info_set(info_set, "psnam_libnam_allocation_id", getenv("SLURM_NAM_ALLOC_ID");
MPI_Info_set(info_set, "psnam_structure", "psnam_structure_raw_and_flat");

MPI_Win_allocate(allocation_size, 1, info_set, MPI_COMM_WORLD, NULL, &raw_nam_win);
MPI_Win_get_info(raw_nam_win, &info_get);
MPI_Info_get(info_get, "psnam_allocation_id", MPI_MAX_INFO_VAL, segment_name, &flag);

MPI_Info_set(info_set, "psnam_manifestation", "psnam_manifestation_segment");
MPI_Info_set(info_set, "psnam_segment_allocation_id", segment_name);
sprintf(offset_value_str, "%d", (allocation_size / num_ranks) * my_rank);
MPI_Info_set(info_set, "psnam_segment_offset", offset_value_str);

MPI_Info_set(info_set, "psnam_structure", "psnam_structure_managed_contiguous");
MPI_Win_allocate(num_int_elements * sizeof(int), sizeof(int), info_set, MPI_COMM_WORLD, NULL, &win);
…
```

Accessing Data in NAM Memory

Accesses to the NAM memory must always be made via MPI_PUT and MPI_GET calls. Direct load/store accesses are (of course) not possible—and MPI_ACCUMULATE is currently also *not* supported since the NAM is just a passive memory device, at least so far. However, after an epoch of accessing the NAM, the respective origin buffers must not be reused or read until a synchronization has been performed. Currently, only the MPI_WIN_FENCE mechanism is supported for doing so. According to this loosely-synchronous model, computation phases alternate with NAM access phases, each completed by a call of MPI_WIN_FENCE, acting as a memory barrier and process synchronization point.

Example

```
for (pos = 0; pos < region_size; pos++) put_buf[pos] = (int)(put_rank+pos);
MPI_Put(put_buf, region_size, MPI_INT, target_region_rank, 0, region_size, MPI_INT, win);
MPI_Get(get_buf, region_size, MPI_INT, target_region_rank, 0, region_size, MPI_INT, win);

MPI_Win_fence(0, win);

for (pos = 0; pos < region_size - WIN_DISP; pos++) {
        if (get_buf[pos] != (int(put_rank+pos) {
                fprintf(stderr, "ERROR at %d: %d vs. %d\n", pos, (int)get_buf[pos], put_rank+pos);

MPI_Win_fence(0, win);
…
```

Alternative interface

The extensions presented so far were all of semantic nature, i.e. without introducing new API functions. However, the changed usage of MPI standard functions may also be a bit confusing, which is why a set of macros is also provided, which in turn encapsulate the MPI functions used for the NAM handling. That way, readability of application code with NAM employment can be improved. These encapsulating macros are the following:

- MPIX_Win_allocate_intercomm(size, disp_unit, info_set, comm, intercomm, win)
  …alias for MPI_Win_allocate()

- MPIX_Win_connect_intercomm(window_name, info, root, comm, intercomm)
  …alias for MPI_Comm_connect()

- MPIX_Win_create_intercomm (info, comm, win)
  …alias for MPI_Win_create_dynamic()

- MPIX_Win_intercomm_query(win, rank, size, disp_unit)
  …alias for MPI_Win_shared_query()