Wikiprint Book

Title: Public/ParaStationMPI

Subject: DEEP - Public/ParaStationMPI

Version: 36

Date: 16.05.2024 00:54:14

Table of Contents

…a modularity-enabled MPI library.

Modular MPI Jobs

Inter-module MPI Communication

ParaStation MPI provides support for inter-module communication in federated high-speed networks. Therefore, so-called Gateway (GW) daemons bridge the MPI traffic between the modules. This mechanism is transparent to the MPI application, i.e., the MPI ranks see a common `MPI_COMM_WORLD` across all modules within the job. However, the user has to account for these additional Gateway resources during the job submission. The following `srun` command line with so-called *colon notation* illustrates the submission of heterogeneous pack jobs including the allocation of Gateway resources:

```
srun --gw_num=1 --partition dp-cn -N8 -n64 ./mpi_hello : --partition dp-esb -N16 -n256 ./mpi_hello
```

An MPI job started with this colon notation via srun will run in a single `MPI_COMM_WORLD`.

However, workflows across modules may demand for multiple `MPI_COMM_WOLRD` sessions that may connect (and later disconnect) with each other during runtime. The following simple job script is example that supports such a case:

```
#!/bin/bash
#SBATCH --gw_num=1
#SBATCH --nodes=8 --partition=dp-cn
#SBATCH hetjob
#SBATCH --nodes=16 --partition=dp-esb

srun -n64  --het-group 0 ./mpi_hello_accept &
srun -n256 --het-group 1 ./mpi_hello_connect &
wait
```

Further examples of Slurm batch scripts illustrating the allocation of heterogeneous resources can be found here.

Application-dependent Tuning

The Gateway protocol supports the fragmentation of larger messages into smaller chunks of a given length, i.e., the Maximum Transfer Unit (MTU). This way, the Gateway daemon may benefit from pipelining effect resulting in an overlapping of the message transfer from the source to the Gateway daemon and from the Gateway daemon to the destination. The chunk size may be influenced by setting the following environment variable:

```
PSP_GW_MTU=<chunk size in byte>
```

The optimal chunk size is highly dependent on the communication pattern and therefore has to be chosen for each application individually.

API Extensions for MSA awareness

Besides transparent MSA support, there is the possibility for the application to adapt to modularity explicitly.

For doing so, on the one hand, ParaStation MPI provides a portable API addition for retrieving topology information by querying a *Module ID* via the `MPI_INFO_ENV` object:

```
 int module_id;
char value[MPI_MAX_INFO_VAL];

MPI_Info_get(MPI_INFO_ENV, "msa_module_id", MPI_MAX_INFO_VAL, value, &flag);

if (flag) { /* This MPI environment is modularity-aware! */

  my_module_id = atoi(value); /* Determine the module affinity of this process. */

} else { /* This MPI environment is NOT modularity-aware! */

        my_module_id = 0;        /* Assume a flat topology for all processes. */
}
```

On the other hand, there is the possibility to use a newly added *split type* for the standardized `MPI_Comm_split_type()` function for creating MPI communicators according to the modular topology of an MSA system:

```
  MPI_Comm_split(MPI_COMM_WORLD, my_module_id, 0, &module_local_comm);

 /*    After the split call, module_local_comm contains from the view of each
  *    process all the other processes that belong to the same local MSA module.
  */

 MPI_Comm_rank(module_local_comm, &my_module_local_rank);

 printf("My module ID is %d and my module-local rank is %d\n", my_module_id, my_module_local_rank);
```

Reporting of Statistical Information

The recently installed **ParaStation MPI version 5.4.7-1** offers the possibility to collect statistical information and to print a respective report on the number of messages and the distribution over their length at the end of an MPI run. (The so-called psmpi **histogram** feature.) This new feature is currently enabled on DEEP-EST for the psmpi installation in the `Devel-2019a` stage:

```
> module use $OTHERSTAGES
> module load Stages/Devel-2019a
> module load GCC/8.3.0
> module load ParaStationMPI/5.4.7-1
```

For activating this feature for an MPI run, the `PSP_HISTOGRAM=1` environment variable has to be set:

```
> PSP_HISTOGRAM=1 srun --gw_num=1 -A deep --partition=dp-cn -N2 -n2 ./IMB-MPI1 Bcast -npmin 4 : --partition=dp-dam-ext -N2

srun: psgw: requesting 1 gateway nodes
srun: job 101384 queued and waiting for resources
srun: job 101384 has been allocated resources
#------------------------------------------------------------
#    Intel(R) MPI Benchmarks 2019 Update 5, MPI-1 part
#------------------------------------------------------------
...

#----------------------------------------------------------------
# Benchmarking Bcast
# #processes = 4
#----------------------------------------------------------------
      #bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
           0         1000         0.03         0.04         0.04
           1         1000         0.81         6.70         2.81
           2         1000         0.86         6.79         2.69
           4         1000         0.84         6.79         2.69
           8         1000         0.86         6.80         2.72
          16         1000         0.85         6.76         2.68
          32         1000         0.87         6.88         2.67
          64         1000         0.95         7.43         3.38
         128         1000         0.98         7.02         3.18
         256         1000         0.91         8.11         3.68
         512         1000         0.91        10.46         4.80
        1024         1000         1.01        11.13         5.59
        2048         1000         1.07        11.91         6.12
        4096         1000         1.35        12.77         6.78
        8192         1000         1.77        14.81         8.23
       16384         1000         3.24        18.66        11.19
       32768         1000         4.93        25.96        16.14
       65536          640        30.06        38.71        34.03
      131072          320        44.85        60.80        52.53
```

```
      262144          160        66.28       100.63        83.20
      524288           80       109.16       180.59       144.57
     1048576           40       199.61       343.00       271.12
     2097152           20       377.66       666.27       521.72
     4194304           10       736.83      1314.28      1025.35


# All processes entering MPI_Finalize

    bin  freq
     64  353913
    128  6303
    256  6303
    512  6303
   1024  6311
   2048  6303
   4096  6303
   8192  6303
  16384  6303
  32768  6303
  65536  4035
 131072  2019
 262144  1011
 524288  507
1048576  255
2097152  129
4194304  66
8388608  0
16777216  0
33554432  0
67108864  0
```

As one can see, the messages being exchanged between all processes of the run are sorted into *bins* according to their message lengths. The number of bins as well as their limits can be adjusted by the following environment variables:

- `PSP_HISTOGRAM_MIN` (default: 64 bytes) Set the lower limit regarding the message size for controlling the number of bins of the histogram.
- `PSP_HISTOGRAM_MAX` (default: 64 MByte) Set the upper limit regarding the message size for controlling the number of bins of the histogram.
- `PSP_HISTOGRAM_SHIFT` (default: 1 bit position) Set the bit shift regarding the step width for controlling the number of bins of the histogram.

Example:

```
> PSP_HISTOGRAM=1 PSP_HISTOGRAM_SHIFT=2 PSP_HISTOGRAM_MAX=4096 srun --gw_num=1 -A deep --partition=dp-cn -N2 -n2 ./IMB-MPI
...

#---------------------------------------------------
# Benchmarking Barrier
# #processes = 4
#---------------------------------------------------
#repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
        1000         5.02         5.02         5.02


# All processes entering MPI_Finalize

bin  freq
 64  16942
256  0
1024  8
4096  0
```

In this example, 16942 messages were smaller than or equal to 64 Byte of MPI payload, while 8 messages were greater than 256 Byte but smaller than or equal to 1024 Byte.

Please note at this point that all messages larger than `PSP_HISTOGRAM_MAX` are *as well counted* and always fall into the *last bin*. Therefore, in this example, no message of the whole run was larger than 1024 Byte, because the last bin, labeled with 4096 but collecting all messages larger than 1024, is empty.

Filtering by Connection Type

An addition that could make this feature quite useful for statistical analysis in the DEEP-EST project is the fact that the message counters can be filtered by connection types by setting the `PSP_HISTOGRAM_CONTYPE` variable. For example, in the following run, only messages that cross the Gateway are recorded:

```
> PSP_HISTOGRAM_CONTYPE=gw PSP_HISTOGRAM=1 PSP_HISTOGRAM_SHIFT=2 PSP_HISTOGRAM_MAX=4096 srun --gw_num=1 -A deep --partitio
...

#----------------------------------------------------
# Benchmarking Barrier
# #processes = 4
#----------------------------------------------------
#repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
        1000         4.96         4.96         4.96


# All processes entering MPI_Finalize

bin  freq (gw)
 64  12694
256  0
1024  4
4096  0
```

Connection types for `PSP_HISTOGRAM_CONTYPE`that might be relevant for DEEP-EST are:

- `gw` for messages routed via a Gateway
- `openib` for InfiniBand communication via the pscom4openib plugin
- `velo` for Extoll communication via the pscom4velo plugin
- `shm` for node-local communication via shared-memory.

A note on performance impacts

The collection of statistical data generates a small overhead, which may be reflected in the message latencies in particular. It is therefore recommended to set `PSP_HISTOGRAM=0` for performance benchmarking — or even better to use another psmpi version and/or installation where this feature is already disabled at compile time.

Modularity-aware Collectives

Feature Description

In the context of DEEP-EST and MSA, ParaStation MPI has been extended by modularity awareness also for collective MPI operations. In doing so, an MSA-aware collective operation is conducted in a hierarchical manner where the intra- and inter- module phases are strictly separated:

i. First do all module-internal gathering and/or reduction operations if required.
ii. Then perform the inter-module operation with only one process per module being involved.
iii. Finally, distribute the data within each module in a strictly module-local manner.

This approach is here exemplarily shown in the following figure for a Broadcast operation with nine processes and three modules:

Besides Broadcast, the following collective operations are currently provided with this awareness:

- `MPI_Bcast` / `MPI_Ibcast`

- `MPI_Reduce` / `MPI_Ireduce`
- `MPI_Allreduce` / `MPI_Iallreduce`
- `MPI_Scan` / `MPI_Iscan`
- `MPI_Barrier`

Feature usage on the DEEP-EST prototype

For activating/controlling this feature, the following environment variables must/can be used:

```
- PSP_MSA_AWARENESS=1 # Generally activate the consideration of modular topologies (NOT enabled by default)
- PSP_MSA_AWARE_COLLOPS=0|1|2 # Select the feature level:
  0: Disable MSA awareness for collective MPI operations
  1: Enable MSA awareness for collective MPI operations (default if PSP_MSA_AWARENESS=1 is set)
  2: Apply MSA awareness recursively in multi-level topologies (set PSP_SMP_AWARENESS=1 in addition)
```

In the recursive application of MSA awareness (`PSP_MSA_AWARE_COLLOPS=2`), a distinction is first made between inter- and intra-**module** communication and then, in a second step, likewise between inter- and intra-**node** communication within the modules if `PSP_SMP_AWARENESS=1` is set in addition. (Please note that a meaningful usage of `PSP_MSA_AWARE_COLLOPS=2` requires `psmpi-5.4.5` or higher.)

A larger benchmarking campaign concerning the benefits of the MSA-aware collectives on the DEEP-EST prototype is still to be conducted. However, by using the histogram feature of ParaStation MPI, it could at least be proven that the number of messages crossing a Gateway can actually be reduced.

Feature usage in environments without MSA support

On the DEEP-EST prototype, the Module ID is determined automatically and the environment variable `PSP_MSA_MODULE_ID` is then set accordingly. However, on systems without this support, and/or on systems with a ParaStation MPI *before* version 5.4.6, the user has to set and pass this variable explicitly, for example, via a bash script:

```
> cat msa.sh
#!/bin/bash
ID=$2
APP=$1
shift
shift
ARGS=$@
PSP_MSA_MODULE_ID=${ID} ${APP} ${ARGS}

> srun ./msa.sh ./IMB-MPI1 Bcast : ./msa.sh ./IMB-MPI1 Bcast
```

In addition, this script approach can always be useful if the user wants to set the Module IDs explicitly, e.g. for debugging and/or emulating reasons.

CUDA Support by ParaStation MPI

CUDA awareness for MPI

In brief, ?CUDA awareness in an MPI library means that a mixed CUDA + MPI application is allowed to pass pointers to CUDA buffers (these are memory regions located on the GPU, the so-called *Device* memory) directly to MPI functions such as `MPI_Send()` or `MPI_Recv()`. A non CUDA-aware MPI library would fail in such a case because the CUDA-memory cannot be accessed directly, e.g., via load/store or `memcpy()` but has to be transferred in advance to the host memory via special routines such as `cudaMemcpy()`. As opposed to this, a CUDA-aware MPI library recognizes that a pointer is associated with a buffer within the device memory and can then copy this buffer prior to the communication into a temporarily host buffer — what is called *staging* of this buffer. Additionally, a CUDA-aware MPI library may also apply some kind of optimizations, e.g., by means of exploiting so-called *GPUDirect* capabilities that allow for direct RDMA transfers from and to the device memory.

Usage on the DEEP-EST system

On the DEEP-EST system, the CUDA awareness can be enabled by loading a module that links to a dedicated ParaStation MPI library providing CUDA support:

```
module load GCC
module load ParaStationMPI/5.4.2-1-CUDA
```

Please note that CUDA awareness might impact the MPI performance on systems parts where CUDA is not used. Therefore, it might be useful (and the other way around necessary) to disable/enable the CUDA awareness. Furthermore, additional optimizations such as GPUDirect, i.e., direct RMA transfers to/from CUDA device memory, are available with certain pscom plugins depending on the underlying hardware.

The following environment variables may be used to influence the CUDA awareness in ParaStation MPI

```
PSP_CUDA=0|1 # disable/enable CUDA awareness
PSP_UCP=1    # support GPUDirect via UCX in InfiniBand networks (e.g., this is currently true for the ESB nodes)
```

Testing for CUDA awareness

ParaStation MPI features three API extensions for querying whether the MPI library at hand is CUDA-aware or not.

The first targets the compile time:

```
#if defined(MPIX_CUDA_AWARE_SUPPORT) && MPIX_CUDA_AWARE_SUPPORT
printf("The MPI library is CUDA-aware\n");
#endif
```

…and the other two also the runtime:

```
if (MPIX_Query_cuda_support())
   printf("The CUDA awareness is activated\n");
```

or alternatively:

```
MPI_Info_get(MPI_INFO_ENV, "cuda_aware", ..., value, &flag);
/*
* If flag is set, then the library was built with CUDA support.
* If, in addition, value points to the string "true", then the
* CUDA awareness is also activated (i.e., PSP_CUDA=1 is set).
*/
```

Please note that the first two API extensions are similar to those that Open MPI also provides with respect to CUDA awareness, whereas the latter is specific solely to ParaStation MPI, but which is still quite portable due to the use of the generic `MPI_INFO_ENV` object.

Using Network Attached Memory with ParaStation MPI

Documentation

- Proposal for accessing the NAM via MPI
- This Manual for using the NAM as a PDF

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.

Attention: *Currently (as of May 2021), changes to the DEEP-EST system are foreseeable, which will also affect the availability of libNAM and the SW-NAM mockup. The following text still reflects the current state and will soon be adapted accordingly.*

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_manifestation_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.

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 MPI Windows

General Semantics

A central use-case for the NAM in DEEP-EST is 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 PSNAM 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 Memory 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    // intra-communicator over which call is collective (handle)
OUT newcomm //inter-communicator 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
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 Memory and Segments

Without further info parameters than described so far, \texttt{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 Use of Segments

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] = 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] != put_rank+pos) {
       fprintf(stderr, "ERROR at %d: %d vs. %d\n", pos, get_buf[pos], put_rank+pos);
               }
}
```

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)` ...as an alias for `MPI_Win_allocate()`.
- `MPIX_Win_connect_intercomm(window_name, info, root, comm, intercomm)` ...as an alias for `MPI_Comm_connect()`.
- `MPIX_Win_create_intercomm(info, comm, win)` ...as an alias for `MPI_Win_create_dynamic()`.
- `MPIX_Win_intercomm_query(win, rank, size, disp_unit)` ...as an alias for `MPI_Win_shared_query()`.