Table of Contents

…a modularity-enabled MPI library.

Heterogeneous Jobs using 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`s 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 GW protocol supports the fragmentation of larger messages into smaller chunks of a given length, i.e., the Maximum Transfer Unit (MTU). This way, the GW daemon may benefit from pipelining effect resulting in an overlapping of the message transfer from the source to the GW daemon and from the GW 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.

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
```

For psmpi versions *before* 5.4.6, the Module IDs (`PSP_MSA_MODULE_ID`) were not set automatically! This means that the user had to set and pass this variable explicitly, for example, via a bash script:

```
#!/bin/bash
# Script (script0.sh) for Module 0: (e.g. Cluster)
APP="./IMB-MPI1 Bcast"
export PSP_MSA_AWARENESS=1
export PSP_MSA_MODULE_ID=0  # <- set an arbitrary ID for this module!
./${APP}
```

```
#!/bin/bash
# Script (script1.sh) for Module 1: (e.g. ESB)
APP="./IMB-MPI1 Bcast"
export PSP_MSA_AWARENESS=1
export PSP_MSA_MODULE_ID=1 # <- set a different ID for this module!
./${APP}
```

```
> srun ./script0 : ./script1
```

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

CUDA Support by ParaStation MPI

What is 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.

Some external Resources

- ?Getting started with CUDA (by NVIDIA)
- ?NVIDIA GPUDirect Overview (by NVIDIA)
- ?Introduction to CUDA-Aware MPI (by NVIDIA)

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)
```

NAM Integration for ParaStation MPI

Documentation

- Proposal for accessing the NAM via MPI
- API Prototype Implementation
- Usage Example on the DEEP-EST SDV

API Prototype Implementation

For evaluating the proposed semantics and API extensions, we have already developed a shared-memory-based prototype implementation where the persistent NAM is (more or less) ?emulated? by persistent shared-memory (with *deep_mem_kind=deep_mem_persistent*).

Advice to users

Please note that this prototype is not intended to actually emulate the NAM but shall rather offer a possibility for the later users and programmers to evaluate the proposed semantics from the MPI application?s point of view. Therefore, the focus here is not put on the question of how remote memory is managed at its location (currently by MPI processes running local to the memory later by the NAM manager or the NAM itself), but on the question of how process-foreign memory regions can be exposed locally. That means that (at least currently) for accessing a persistent RMA window, it has to be made sure that there is at least one MPI process running locally to each of the window?s memory regions.

Extensions to MPI

The API proposal strives to stick to the current MPI standard as close as possible and to avoid the addition of new API functions and other symbols. However, in order to make the usage of the prototype a little bit more convenient for the user, we have added at least a small set of new symbols (denoted with MPIX) that may be used by the applications.

```
extern int MPIX_WIN_DISP_UNITS;
#define MPIX_WIN_FLAVOR_INTERCOMM        (MPI_WIN_FLAVOR_CREATE +        \
                                         MPI_WIN_FLAVOR_ALLOCATE +        \
                                         MPI_WIN_FLAVOR_DYNAMIC +        \
                                         MPI_WIN_FLAVOR_SHARED + 0)
#define MPIX_WIN_FLAVOR_INTERCOMM_SHARED (MPI_WIN_FLAVOR_CREATE +        \
                                         MPI_WIN_FLAVOR_ALLOCATE +        \
                                         MPI_WIN_FLAVOR_DYNAMIC +        \
                                         MPI_WIN_FLAVOR_SHARED + 1)
```

Code Example for a ?Hello World? workflow

The following two C codes should demonstrate how it shall become possible to pass intermediate data between two subsequent steps of a workflow (Step 1: hello / Step 2: world) via the persistent memory of the NAM (currently emulated by persistent shared-memory):

```
/** hello.c **/

    /* Create persistent MPI RMA window: */
       MPI_Info_create(&win_info);
       MPI_Info_set(win_info, "deep_mem_kind", "deep_mem_persistent");
       MPI_Win_allocate(sizeof(char) * HELLO_STR_LEN, sizeof(char), win_info, MPI_COMM_WORLD,
                                         &win_base, &win);

       /* Put some content into the local region of the window: */
       if(argc > 1) {
               snprintf(win_base, HELLO_STR_LEN, "Hello World from rank %d! %s", world_rank, argv[1]);
       } else {
               snprintf(win_base, HELLO_STR_LEN, "Hello World from rank %d!", world_rank);
       }
```

```
        MPI_Win_fence(0, win);

        /* Retrieve port name of window: */
        MPI_Info_free(&win_info);
        MPI_Win_get_info(win, &win_info);
        MPI_Info_get(win_info, "deep_win_port_name", INFO_VALUE_LEN, info_value, &flag);

        if(flag) {
                strcpy(port_name, info_value);
                if(world_rank == root) printf("(%d) The Window's port name is: %s\n", world_rank, port_name);
        } else {
                if(world_rank == root) printf("(%d) No port name found!\n", world_rank);
        }
```

```
/** world.c **/

        /* Check for port name: (to be passed as a command line argument) */
        if(argc == 1) {
                if(world_rank == root) printf("[%d] No port name found!\n", world_rank);
                goto finalize;
        } else {
                strcpy(port_name, argv[1]);
                if(world_rank == root) printf("[%d] The Window's port name is: %s\n", world_rank, port_name);
        }

        /* Try to connect to the persistent window: */
        MPI_Info_create(&win_info);
        MPI_Info_set(win_info, "deep_win_connect", "true");
        MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
        errcode = MPI_Comm_connect(port_name, win_info, root, MPI_COMM_WORLD, &inter_comm);
        printf("[%d] Connection to persistent memory region established!\n", world_rank);

         /* Retrieve the number of remote regions: (= former number of ranks) */
        MPI_Comm_remote_size(inter_comm, &remote_size);
        if(world_rank == root) printf("[%d] Number of remote regions: %d\n", world_rank, remote_size);

        /* Create window object for accessing the remote regions: */
        MPI_Win_create_dynamic(MPI_INFO_NULL, inter_comm, &win);
        MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_flavor, &flag);
        assert(*create_flavor == MPIX_WIN_FLAVOR_INTERCOMM);
        MPI_Win_fence(0, win);

        /* Check the accessibility and the content of the remote regions: */
        for(i=0; i<remote_size; i++) {
                char hello_string[HELLO_STR_LEN];
                MPI_Get(hello_string, HELLO_STR_LEN, MPI_CHAR, i, 0, HELLO_STR_LEN, MPI_CHAR, win);
                MPI_Win_fence(0, win);
                printf("[%d] Get from %d: %s\n", world_rank, i, hello_string);
        }
```

Usage Example on the DEEP-ER SDV

On the DEEP-ER SDV, there is already a special version of ParaStation MPI installed that features all the introduced API extensions. It is accessible via the module system:

```
> module load parastation/5.2.1-1-mt-wp6
```

When allocating a session with N nodes, one can run an MPI session (let?s say with n processes distributed across the N nodes) where each of the processes is contributing its local and persistent memory region to an MPI window:

```
> salloc --partition=sdv --nodes=4 --time=01:00:00
salloc: Granted job allocation 2514
> srun -n4 -N4 ./hello 'Have fun!'
(0) Running on deeper-sdv13
(1) Running on deeper-sdv14
(2) Running on deeper-sdv15
(3) Running on deeper-sdv16
(0) The Window's port name is: shmid:347897856:92010569
(0) Calling finalize...
(1) Calling finalize...
(2) Calling finalize...
(3) Calling finalize...
(0) Calling finalize...
(0) Finalize done!
(1) Finalize done!
(2) Finalize done!
(3) Finalize done!
```

Afterwards, on all the nodes involved (and later on the NAM) one persistent memory region has been created by each of the MPI processes. The ?port name? for accessing the persistent window again is in this example:

```
shmid:347897856:92010569
```

By means of this port name (here to be passes as a command line argument), all the processes of a subsequent MPI session can access the persistent window provided that there is again at least one MPI processes running locally to each of the persistent but distributed regions:

```
> srun -n4 -N4 ./world shmid:347897856:92010569
[0] Running on deeper-sdv13
[1] Running on deeper-sdv14
[2] Running on deeper-sdv15
[3] Running on deeper-sdv16
[0] The Window's port name is: shmid:347897856:92010569
[1] Connection to persistent memory region established!
[3] Connection to persistent memory region established!
[0] Connection to persistent memory region established!
[2] Connection to persistent memory region established!
[0] Number of remote regions: 4
[0] Get from 0: Hello World from rank 0! Have fun!
[1] Get from 0: Hello World from rank 0! Have fun!
[2] Get from 0: Hello World from rank 0! Have fun!
[3] Get from 0: Hello World from rank 0! Have fun!
[0] Get from 1: Hello World from rank 1! Have fun!
[1] Get from 1: Hello World from rank 1! Have fun!
[2] Get from 1: Hello World from rank 1! Have fun!
[3] Get from 1: Hello World from rank 1! Have fun!
[0] Get from 2: Hello World from rank 2! Have fun!
[1] Get from 2: Hello World from rank 2! Have fun!
[2] Get from 2: Hello World from rank 2! Have fun!
[3] Get from 2: Hello World from rank 2! Have fun!
[0] Get from 3: Hello World from rank 3! Have fun!
[1] Get from 3: Hello World from rank 3! Have fun!
[2] Get from 3: Hello World from rank 3! Have fun!
[3] Get from 3: Hello World from rank 3! Have fun!
[0] Calling finalize...
[1] Calling finalize...
[2] Calling finalize...
[3] Calling finalize...
[0] Finalize done!
[1] Finalize done!
[2] Finalize done!
```

```
[3] Finalize done!
```

Advice to users

Pleases note that if not all persistent memory regions are covered by the subsequent session, the ?connection establishment? to the remote RMA window fails:

```
> srun -n4 -N2 ./world shmid: 347897856:92010569
[3] Running on deeper-sdv14
[1] Running on deeper-sdv13
[2] Running on deeper-sdv14
[0] Running on deeper-sdv13
[0] The Window's port name is: shmid:347930624:92010569
[0] ERROR: Could not connect to persistent memory region!
application called MPI_Abort(MPI_COMM_WORLD, -1)
?
```

Cleaning up of persistent memory regions

If the connection to a persistent memory region succeeds, the window and all of its memory regions will eventually be removed by the MPI_Win_free call of the subsequent MPI session (here by world.c) at least if not deep_mem_persistent is passes again as an Info argument. However, if a connection attempt fails, the persistent memory regions still persist.

For explicitly cleaning up those artefacts, one can use a simple batch script:

```
#!/bin/bash
keys=`ipcs | grep 777 | cut -d' ' -f2`
for key in $keys ; do
       ipcrm -m $key
done
```

Advice to administrators

Obviously, a good idea would be the integration of e such an automated cleaning-up procedure as a default into the epilogue scripts for the jobs.