Split_types issues

Guillaume Mercier/Inria TADaaM Team

1 Context and issues

During the last MPI Forum Meeting in Chattanooga (late Feb. 2019), some questions were raised about the relevance of :

— standardizing a set of `split_types` values for `MPI_Comm_split_type`
— creating query mechanisms that would allow the user to know the set of types avaialble on the target architecture (standard or non-standard types alike)
— other things ?

This context is the following :

— the hierarchical communicator proposal relies on the introduction of a new `split_type` value for `MPI_Comm_split_type` : `MPI_COMM_TYPE_HW_TOPOLOGY/ MPI_COMM_TYPE_HW_DOMAIN/MPI_COMM_TYPE_HARDWARE/whatever)` indicating unambiguously that the splitting of the input communicator is based on hardware topology criteria. There `info` argument can be used to steer the splitting action, but not necessarily. If no info is provided, then a recursive splitting is performed, based on the hierarchical nature of the hardware and the input communicator.
— in the Open MPI implementation of `MPI_Comm_split_type`, a large set of `split_type` values are supported (e.g. `OMPI_COMM_TYPE_CORE`, `OMPI_COMM_TYPE_NUMA`, `OMPI_COMM_TYPE_L3CACHE`, etc.) that are based on the types available in hwloc.
— the MPICH implementation of `MPI_Comm_split_type` supports the `split_type` value `MPIX_COMM_TYPE_NEIGHBORHOOD` to perform a hardware topology-aware splitting. The `info` argument is also used to decide at which level/resource the operation is to be performed. hwloc strings are used for this purpose.
— Rolf Rabenseifner's proposal (issue #120) introduces a set of new functions that optimize the creation of cartesian topologies by taking into account the hardware topology factor. Some of the proposed functions could use the "hardware communicators" created with `MPI_Comm_split_type`. some other require the use of new `split_type` values.

2 About the hierarchical communicators

2.1 Summary of the proposal

The hierarchical communicators proposal is based on the introduction of a new `split_type` value (`MPI_COMM_TYPE_HW_DOMAIN` in the rest of this document) which indicates clearly that the criterion to produce new subcommunicators is hardware topology-based. This new version of `MPI_Comm_split_type` :

— Splits the input communicator into subcommunicators, each of which encompasses MPI processes that do share resources in the underlying physical topology (e.g. a network switch, a physical node, a L3 cache, a L2 cache, a core, etc.). This sharing of resources is detected or known by the *implementation* (its process manager for instance) or another subsystem (e.g hwloc).

— May use a `MPI_HW_DOMAIN_TYPE` *keyval* : If this key is defined (in the `info` parameter passed to the function), `MPI_COMM_SPLIT_TYPE` can then perform a split operation on a *specific hardware level*, whose name is the value of the `MPI_HW_DOMAIN_TYPE` key. **The levels names are not standardized and therefore implementation-dependent. Such names should be queried by a new, dedicated function (see later)**. For instance, an HWLOC-based implementation could use the HWLOC types names as *domain* types :
  — `HWLOC_OBJ_MACHINE`
  — `HWLOC_OBJ_PACKAGE`
  — `HWLOC_OBJ_CORE`
  — `HWLOC_OBJ_PU` for hardware threads
  — `HWLOC_OBJ_NUMANODE`
  — `HWLOC_OBJ_L1CACHE`, ..., `HWLOC_OBJ_L5CACHE`
  — `HWLOC_OBJ_GROUP` for other hierarchy levels

  Simpler (shorter) names (e.g. `Core`, `Socket`, `Machine`) could (should ?) also be used. **In this way, names are still not specified in the MPI standard but can be used nevertheless. Despite that the names are not standard, the *way* of accessing a hardware level (that is, the sequence of MPI calls to achieve this) is standard and the same, regardless of the implementation and the names they internally define and use.**

2.2 Implementation guidelines

An implementation can support (implement) this addition to `MPI_COMM_SPLIT_TYPE` in multiple ways which are all acceptable with regard to the defined behaviour (i.e. new communicators materialize the sharing of physical resources between processes) :

— If no valid communicator can be created, the value `MPI_COMM_NULL` is returned, and this value can be used to assess if the last level of the hierarchy has been reached. In particular, it is possible for an implementation to not produce subcommunicators by directly returning the value.

— An implementation can simply return `MPI_COMM_NULL` if it cannot/does not want to support this feature.

— An implementation can choose to create a new communicator for *every* level present in the underlying physical hierarchy.

— Alternatively, in order to avoid the creation of redundant objects, the implementation can decide that the group of MPI processes supporting a new subcommunicator can be a *strict* subset of the group supporting the input (parent) communicator. More specifically, a call to `MPI_COMM_COMPARE(comm,newcomm)` should return `MPI_UNEQUAL` in this case. In case of several possible levels, the implementation is free to return the level considered as the most relevant, for instance the *highest* or the *lowest* level in the hierarchy.

2.3 Usage

There are two ways to use this function, the *unguided* mode and the *guided* mode.

— **With the *unguided* mode**, it is possible to capture the hierarchical nature of the underlying hardware by calling "recursively" `MPI_COMM_SPLIT_TYPE` with `MPI_COMM_TYPE_HW_DOMAIN` as the `split_type` value on newly created subcommunicators and *no info* is provided to guide the split operation. Code example :

```
  #define NLEVELS 64 /* random value */

  MPI_Comm newcomm[NLEVELS];
  MPI_Comm oldcomm = MPI_COMM_WORLD;
  int rank, idx = 0;

  while((oldcomm != MPI_COMM_NULL) && (idx < NLEVELS))
  {
    MPI_Comm_rank(oldcomm,&rank);
    MPI_Comm_split_type(oldcomm,
                        MPI_COMM_TYPE_HW_DOMAIN,
                        rank,
                        MPI_INFO_NULL,
                        &newcomm[idx]);
    oldcomm = newcomm[idx++];
  }
```

— **With the *guided* mode**, an info keyval has also to be provided (along with the `MPI_COMM_TYPE_HW_DOMAIN` `split_type` value) in order to create a communicator corresponding to a *specific* hardware level in the hierarchy. The key name is `MPI_HW_DOMAIN_TYPE` and **its value should be a level name**. In order to obtain information about the physical hierarchy, that is, the number of levels exposed by the implementation (levels being redundant or not) as well as **the various level names** (values for standard-defined keys) the query function `MPIX_GET_HW_TOPOLOGY_INFO` must be called (see later. This function uses keyvals whose names are `MPI_HW_LEVEL0`,. . .,`MPI_HW_LEVELN`). Code example :

```
{
 MPI_Comm out_comm;
 MPI_Comm oldcomm = MPI_COMM_WORLD;
 MPI_Info info;
 int rank, idx, flag;
 char *resource_type = NULL;
 char str[MPI_MAX_INFO_VAL+1];
 char str2[MPI_MAX_INFO_VAL+1];
 char domain_names[64][MPI_MAX_DOMAIN_NAME+1];
 int names_len[64];

 MPI_Comm_rank(oldcomm,&rank);
 MPI_Info_create(&info);

 /* Query  hardware details */

 /* First possible function */
#idef FIRST
 MPIX_Get_hw_topology_info(&numlevels,info);
#else
 /* Other posssibility */
 MPIX_Get_hardware_names(domain_names,names_len) ;
#endif

 fprintf(stdout,"Number of levels available: %i\n",numlevels);

 /* could be retrieved with MPI_Info_get_nthkey instead */
 for(idx = 0 ; idx < numlevels; idx++){
   sprintf(str,"MPI_HW_LEVEL%d",idx);
   MPI_Info_get(info,str,MPI_MAX_INFO_VAL-1,str2,&flag);
   if (flag)
     fprintf(stdout,"%s type is %s\n",str,str2);
 }

 /* Let us suppose that Package is an available hardware level: */
 /* split at this level                                         */
 MPI_Info_set(info,"MPI_HW_DOMAIN_TYPE","Package");
 MPIX_Comm_split_type(oldcomm,MPI_COMM_TYPE_HW_DOMAIN,rank,info,&out_comm);

 /* Or split at level 2 in the hierarchy */
 /* (whatever its name is)              */
 MPI_Info_get(info,"MPI_HW_LEVEL2",MPI_MAX_INFO_VAL-1,str2,&flag);
 if(flag){
   MPI_Info_set(info,"MPI_HW_DOMAIN_TYPE",str2);
   MPIX_Comm_split_type(oldcomm,MPI_COMM_TYPE_HW_DOMAIN,rank,info,&out_comm);
 }
}
```

2.4 Relationship with the mapping/binding of processes

The mapping/binding of MPI processes should not be assumed when using this splitting mechanisms. It actually is independent from it. However, the mapping and binding of MPI processes onto physical resources must be taken into account for subcommunicators creation. Indeed, the *deepest* hardware level in the hierarchy corresponding to a subcommunicator should always correspond to resource the calling MPI process is bound to. For instance, if a process is bound to a certain cache level, no information below this cache level can be returned, as the MPI process can possibly use any of the caches below the level it is bound to. Any attempt to create a subcommunicator corresponding to a hardware level below the level the MPI process is bound to must return `MPI_COMM_NULL`.

2.5 Query functions

2.5.1 `MPIX_GET_HW_TOPOLOGY_INFO`

`MPIX_GET_HW_TOPOLOGY_INFO(numlevels,info)`

OUT `numlevels` number of levels in the hardware hierarchy (integer)

OUT `info` info object (handle)

C Prototype :

```
int MPIX_Get_hw_topology_info(int *numlevels,MPI_Info info)
```

This function allows the calling MPI process to retrieve information about the underlying hardware topology. Two different pieces of information are available :

— the number of hardware levels in the hierarchy (the `numlevels` parameter)

— a set of keyvals (set in the `info` parameter). There are *numlevels* different keyvals, named `MPI_HW_LEVEL0`, `MPI_HW_LEVEL1`, ... `MPI_HW_LEVEL`*`numlevels-1`*. These values are implementation-defined and are the same as the ones used to define the `MPI_HW_DOMAIN_TYPE` key in `MPIX_COMM_SPLIT_TYPE`.

Another possible design would be to return directly the names (as strings), as in :

```
MPIX_GET_HARDWARE_NAMES(names,names_len)
```

OUT `names` array of strings

OUT `names_len` array of integers

C Prototype :

```
MPI_Get_hardware_names(char **names, /*OUT*/
                       int   *names_len, /*OUT*/);
```

Assuming that the number of levels (e.g. `nlevels`) is known (maybe with another routine), this function takes as its first parameter an array of `nlevels` strings, where `names[i]` represent storage that is at least `MPI_MAX_DOMAIN_NAME` characters long. The function actually writes `names_len[i]` characters plus an additional null character.