Extension of MPI One-Sided Communication API

Piotr Dorożyński and Paweł Czarnul

March 16, 2016

Abstract

This document describes modifications, which will be implemented as wrappers, that need to be applied to MPI one-sided communication functions in order to provide support for persistent memory.

	Project	**Optimized MPI API for persistent memory**
1	Author	Paweł Czarnul
	Version	0.1
	Modification date	December 16, 2014
	Description	Template
2	Author	Piotr Dorożyński
	Version	0.2
	Modification date	December 21, 2014
	Description	Initial description of wrappers to MPI one-sided communication API we need to implement.
3	Author	Piotr Dorożyński and Paweł Czarnul
	Version	0.3
	Modification date	January 15, 2015
	Description	Updates and extensions.
4	Author	Piotr Dorożyński
	Version	0.4
	Modification date	March 16, 2016
	Description	Update after implementation.

1 Introduction

MPI one-sided communication routines enable to specify regions of memory (called windows) of one process that are available for remote read and write for other processes of an MPI

application. This communication normally operates in RAM and as such the usage of persistent memory will not improve its performance (as it is already tuned to operate in a bit faster RAM). It is however possible to benefit from another aspect of persistent memory which is persistence. The main idea of proposed extensions for MPI one-sided communication is to use persistence of persistent memory with support of its performance to apply continuous checkpointing at the code level.

Such checkpointing will be achieved by providing transactions in persistent memory, so that when synchronization occurs and is successful it can be considered as a consistent application state. More details will be provided in Section 5.3, where synchronization schemes are explained.

Proposed extensions will be compatible with MPI semantics defined in MPI standard [1].

2 Important factors and terms

Persistent memory is currently implemented by memory mapping a file on specially prepared file system. As such all changes to persistent memory are immediately visible at the level of the operating system and above, but because of operating system caching the data may not actually be in persistent memory. This means that if there is a need to make sure data is stored the changes need to be flushed. Exact moments when such flushing is needed is described in Section 5.3, where semantics and correctness criteria are described.

In MPI one-sided communication there are two important terms that will be used in this document: *origin* and *target*. In the MPI specification [1] they are defined as:

> We shall denote by **origin** the process that performs the call, and by **target** the process in which the memory is accessed. Thus, in a put operation, source=origin and destination=target; in a get operation, source=target and destination=origin.

3 Persistent memory management design

Persistent memory is presented to users as an additional disk with a specially modified ext4 file system which is mounted into the standard Linux filesystem. To provide automatic management of stored memory windows the implementation needs to know where this filesystem is mounted. Proposed extensions also need to somehow recognize different instances of the same parallel application in order to retrieve regions of persistent memory that are associated with the given instance of an application, when the application is restarted. Both of the above will be achieved by specifying a special function (see Section 7.1) that gets a path to the directory which will be considered as a place where all memory windows will be stored. This also means that application instances will be recognized by their root path.

As mentioned earlier the proposed extensions will provide transactions. The main problem that will be solved by using transactions is that when failure happens processes may

communicate and none of them will know on restart whether this communication has been finished or not. This means that there is a need for definition of moments when all processes have consistent views of memory, so on failure all changes made after such a moment may be rolled back to the previously known consistent global checkpoint. Such moments of consistency are defined by starting and committing a transaction (see Sections 5.3.1 and 5.3.2 for more information). Starting a transaction defines a moment to which the application may roll back and committing a transaction means that now memory is consistent and history of changes since its start may be deleted. Since memory may be modified at any time, each commit must be directly followed by starting a new transaction as otherwise when failure occurs memory may be in unknown state. In the proposed implementation transactions will be performed simply by copying a window area to some other part of persistent memory when a transaction starts and freeing that memory when the transaction is committed. When restarting an application after failure saved data will be copied back to original area effectively rolling back the transaction. Decision for such a mechanism was made, because when memory is modified there is no call to any function, because persistent memory is exposed like RAM, which may be wrapped to save the change.

Since starting a transaction creates a copy of a memory window, these copies may be preserved on commit instead of deleting them, thus resulting in a possibility to restart application from any created checkpoint. Such behavior will be disabled by default and enabled by setting a proper key in `MPI_Info` object (see Section 5.1.6). Created checkpoints will be automatically numbered incrementally and saved under specified names (see Section 5.1.6) on the root path. Checkpoints will be managed by new methods described in Section 7.2 and a window creation method with allocation will provide a method to retrieve a specific version (see Section 5.1.2).

4 Error handling

Error handling will be done as suggested in Section 8.5 of the MPI specification [1]. All errors that may be associated with any standard MPI error class will be associated with this class. All other errors will get their own error codes that will be associated with a newly defined error class. These error codes will be initialized when an application starts, see Section 6 for more information.

Currently the following codes can be defined (additional codes may be defined when needed during the implementation):

`MPI_ERR_PMEM_ROOT_PATH` Invalid root path i.e. root path doesn't exist.

`MPI_ERR_PMEM_NAME` Invalid persistent memory area name i.e. memory area doesn't exist.

`MPI_ERR_PMEM_CKPT_VER` Invalid checkpoint version.

`MPI_ERR_PMEM_MODE` Invalid `mode`.

`MPI_ERR_PMEM_WINDOWS` Invalid `windows` argument.

`MPI_ERR_PMEM_VERSIONS` Invalid `versions` argument.

`MPI_ERR_PMEM_ARG` Invalid argument of some other kind.

`MPI_ERR_PMEM_NO_MEM` Unable to allocate memory.

5 Core MPI one-sided communication API

This section provides information about MPI functions that belong to the one-sided communication API and will need to be modified in order to achieve previously specified goals. Its structure is almost the same as in the MPI specification [1] and for each of the described functions a brief (without going into details that can be found in the specification [1]) explanation is given on what it should do and whether it needs modification or not and why. The implemented wrappers will have names identical to the original MPI functions with `_pmem` suffix. Synchronization functions for MPI one-sided communication doesn't have `MPI_Info` parameter, so it's impossible to provide additional information to it (the `assert` parameter is only a programmer's suggestion that implementation may use and should not be used to extend functionality — asserts should not change function semantics). However, it is necessary to be able to specify when checkpoint should be created and when not, so the following solution was chosen: functions with `_pmem` suffix only unwraps `MPI_Win` structure and calls original MPI functions and functions with `_pmem_persist` provide additional functionality as specified in the next subsections.

5.1 Initialization

5.1.1 Window creation

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
    MPI_Comm comm, MPI_Win *win)
```

This method must be called by all processes in an MPI application. Each process specifies a window of its already allocated memory and gets a handle to a window object used by other operations. Since this method does not allocate any memory no modifications are introduced into this function that would apply to allocating and managing persistent memory. However this method also enables to specify additional information that can be used by the implementation for some optimization. This information is provided in the `info` argument. In proposed extensions this information will be used in order to know whether the specified memory should be treated as persistent memory. This information will not be saved by an underlying MPI implementation (see Section 5.1.6), so it is needed to modify this function just to save this additional information. In the `info` parameter only one key `pmem_is_pmem` is considered by this function. All other keys are simply ignored.

5.1.2 Window that allocates memory

```
int MPI_Win_allocate(MPI_Aint size, int disp_unit, MPI_Info info,
    MPI_Comm comm, void *baseptr, MPI_Win *win)
```

This function is similar to `MPI_Win_create`, but it does not only create a window, but also allocates memory, so it needs modification not only for saving information provided in the `info` parameter, but also to allocate area of persistent memory if specified. This function will also roll back memory to the last consistent state or a provided checkpoint number. All information that controls how memory will be allocated will be provided in the `info` parameter. See Section 5.1.6 for available keys.

5.1.3 Window that allocates shared memory

```
int MPI_Win_allocate_shared(MPI_Aint size, int disp_unit, MPI_Info info,
    MPI_Comm comm, void *baseptr, MPI_Win *win)
```

From the MPI specification [1] one can read that: "`MPI_WIN_ALLOCATE_SHARED` differs from `MPI_WIN_ALLOCATE` in that the allocated memory can be accessed from all processes in the window's group with direct load/store instructions." As no assumptions are made that persistent memory may be used "with direct load/store instructions", it is not needed to modify this function or any other dependent on it. This also means that this mode of operation (persistent memory as shared memory) will not be supported.

5.1.4 Window of dynamically attached memory

```
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

This function works similarly to `MPI_Win_create`, but it does not specify the region of memory that will be exposed to other processes. Similarly to `MPI_Win_create` in the `info` parameter only one key `pmem_is_pmem` is considered. All other keys are simply ignored. Also similarly to `MPI_Win_create` there is a need for saving information provided in the `info` parameter so this function needs to be modified just for this purpose.

For a window created using `MPI_Win_create_dynamic` the following two functions are used to specify memory areas to use:

```
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
```

This function attaches a local memory region to a given window. It is necessary to modify this function to save an address of a specified region in order to use this address for flushing.

```
int MPI_Win_detach(MPI_Win win, const void *base)
```

This function detaches a local memory region from a given window. It is necessary to modify this function to remove a specified region from list of memory regions that it was added to in `MPI_Win_attach`.

5.1.5 Window destruction

```
int MPI_Win_free(MPI_Win *win)
```

This function frees the provided window object. If the provided window was created by a call to `MPI_Win_allocate` then the underlying memory should also be freed. In case of persistent memory it will be done only if optional parameter `pmem_volatile` was set to `true`. It is necessary to modify this function to provide this possibility.

5.1.6 Window info

`MPI_Info` object defines hints that may be used by an implementation to optimize functions in some specific cases. Proposed extensions will use this object to specify how persistent memory will be used by MPI one-sided communication functions. The following keys for `MPI_Info` object will be defined to provide such control:

`pmem_is_pmem` if set to `true` the implementation will assume that an area of memory is in persistent memory, so it will flush changes to persistent memory when needed (see Section 5.3). For `MPI_Win_allocate` it means that an area of memory will be allocated in persistent memory.

`pmem_dont_use_transactions` if set to `true` transactions will be turned off (see Section 5.3). If `pmem_is_pmem` is set to `false` this option is ignored.

`pmem_keep_all_checkpoints` if set to `true` all checkpoints will be preserved so that the user may choose which one he wants to restart from. By default old checkpoints will be deleted when a new consistent state is saved. If `pmem_is_pmem` is set to `false` or `pmem_dont_use_transactions` is set to `true` this option is ignored.

`pmem_name` specifies the name of a persistent memory area to use. The provided name is considered as a name of a file in the filesystem. This name may not contain a path to a file with additional directories. If `pmem_is_pmem` is set to `false` this option is ignored.

`pmem_mode` may be set to `expand` or `checkpoint`. In `expand` mode the specified area will be expanded to the new size and all old checkpoints will be deleted. In `checkpoint` mode the memory area will be filled with data from checkpoint, whose version is specified in `pmem_checkpoint_version` or the newest version if `pmem_checkpoint_version` is not specified. In this mode size must be the same as the size of checkpoint. If `pmem_is_pmem` is set to `false` this option is ignored.

`pmem_checkpoint_version` specifies a checkpoint version to retrieve. If it is not set then the newest version is used. If `pmem_is_pmem` is set to `false` or `pmem_mode` is not set to `checkpoint` this option is ignored.

`pmem_append_checkpoints` if set to `true` new checkpoints will get numbers after the highest checkpoint ever created for this window. Otherwise new checkpoints will get consecutive numbers after `pmem_checkpoint_version` version. If `pmem_is_pmem` is set to `false` or `pmem_mode` is not set to `checkpoint` this option is ignored.

`pmem_global_checkpoint` if set to `true` and `pmem_checkpoint_version` is not set then the value of latest available checkpoint will be determined by finding last consistent checkpoint version across all processes in communicator. If `pmem_is_pmem` is set to `false` or `pmem_mode` is not set to `checkpoint` this option is ignored.

`pmem_volatile` if set to `true` the allocated area will be considered as volatile even though the area of memory is in persistent memory. This means that freeing a window created with this option set to `true` will also free the corresponding area in persistent memory, so it will not be available ever after. Also it stops flushing and providing transactions for such a window. If `pmem_is_pmem` is set to `false` this option is ignored.

The MPI specification [1] defines the following functions to control hints that were supplied in `MPI_Info` object during window creation:

```
int MPI_Win_set_info(MPI_Win win, MPI_Info info)
```

This function changes the set of information provided to window in `MPI_Info` object during window creation. Only two of the specified parameters (`pmem_dont_use_transactions` and `pmem_keep_all_checkpoints`) in the proposed implementation may be changed and this function needs to be modified to provide this possibility.

```
int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
```

According to specification [1]:

> The info object returned in `info_used` will contain all hints currently active for this window. This set of hints may be greater or smaller than the set of hints specified when the window was created, as the system may not recognize some hints set by the user, and may recognize other hints that the user has not set.

This means that the underlying implementation may return only information it recognized and this function needs to be modified to add to those information all hints that were recognized by the proposed implementation.

5.2 Communication calls

All communication methods are based on reading and writing at specified addresses in memory of a specified process. Since persistent memory is exposed to processes like normal RAM it is not needed to modify any of these functions for the purpose of communication. Also

MPI semantics and correctness for one sided communication (see [1] Section 11.7 and Section 5.3 of this document) specify that one can be sure that an operation completed at a target only after an appropriate synchronization call. This means that persistent memory can be flushed only in synchronization calls and thus it is not needed to modify any of communication functions.

5.3 Synchronization calls

MPI defines two modes of synchronization in one-sided communication:

active target communication in which both communicating processes are explicitly involved in the communication by calling appropriate synchronization calls. This kind of communication scenarios will be explained in Sections 5.3.1 and 5.3.2.

passive target communication in which only an origin process is explicitly involved in the communication by calling appropriate synchronization calls. This kind of communication is close to a shared memory model and it is hard to implement effectively when memory is not shared (see *Rationale* from MPI specification [1] below).

> *Rationale.* The implementation of passive target communication when memory is not shared may require an asynchronous software agent. Such an agent can be implemented more easily, and can achieve better performance, if restricted to specially allocated memory. It can be avoided altogether if shared memory is used. It seems natural to impose restrictions that allows one to use shared memory for third party communication in shared memory machines.

As it is not assumed that memory will be shared no support (flushing and transactions) for this kind of communication will be provided.

All one sided communication calls must occur only within one of following epochs:

access epoch is an epoch in which an origin process may invoke communication calls. It starts with an appropriate synchronization call, proceeds with zero or more communication calls and completes with another synchronization call.

exposure epoch is an epoch in which a target window can be accessed. It is started and completed by synchronization calls executed by target process.

Also according to the specification [1]:

> There is a one-to-one matching between access epochs at origin processes and exposure epochs on target processes: RMA [Remote Memory Access] operations issued by an origin process for a target window will access that target window during the same exposure epoch if and only if they were issued during the same access epoch.

MPI specification [1] defines rules when changes made by communication calls have to be visible and they are defined as follows:

> The update performed by a get call in the origin process memory is visible when the get operation is complete at the origin (or earlier); the update performed by a put or accumulate call in the public copy of the target window is visible when the put or accumulate has completed at the target (or earlier).

Above mentioned communication calls completions according to Section 11.7 ("Semantics and Correctness") of MPI specification [1] are only ensured after appropriate synchronization calls are made (see Sections 5.3.1 and 5.3.2 for information about these methods).

All of the above means that only synchronization calls need to be modified in order to provide support for making consistent checkpoints. In fact no information is known about which communication calls have finished on both origin and target until we make a synchronization call.

From the above it may also be seen that only starting or completing an exposure epoch have impact on keeping data consistent, as these are the only moments when one can be sure that all modifications made by all processes have ended. This leads to the conclusion that each start or end of an exposure epoch should flush all the changes to persistent memory and also commit all pending transactions and start new ones. One needs to remember that exposure epoch applies only to remote accesses, but local writes may be done in between and so it is needed to save these too.

By default the implementation will make all one sided communication in pmem transactional, but transactions can be turned off by specifying `pmem_dont_use_transactions` in `MPI_Info` when creating a window (see Section 5.1.2 and 5.1.6). This may be useful when a programmer makes own transactions or uses "double buffering" i.e. results of computations are saved in another location than the source and in each iteration the source and destination memory areas are swapped. In the "double buffering" scenario source memory area is not overwritten until destination is filled, so when restarting after failure the source data is not changed and can be used directly once again.

Now let us be more specific. MPI provides three synchronization mechanisms:

`MPI_Win_fence` is used to provide global synchronization. According to the specification [1]:

> This call is used for active target communication. An access epoch at an origin process or an exposure epoch at a target process are started and completed by calls to `MPI_Win_fence`. A process can access windows at all processes in the group of `win` during such an access epoch, and the local window can be accessed by all processes in the group of `win` during such an exposure epoch.

This synchronization mechanism is described in Section 5.3.1.

`MPI_Win_start, MPI_Win_complete, MPI_Win_post, MPI_Win_wait` are used for restricting synchronization to the minimum, so only pairs of communicating processes synchronize. According to the specification [1]:

> These calls are used for active target communication. An access epoch is started at the origin process by a call to `MPI_Win_start` and is terminated by a call to `MPI_Win_complete`. The start call has a group argument that specifies the group of target processes for that epoch. An exposure epoch is started at the target process by a call to `MPI_Win_post` and is completed by a call to `MPI_Win_wait`. The post call has a group argument that specifies the set of origin processes for that epoch.

This synchronization mechanism is described in Section 5.3.2.

`MPI_Win_lock, MPI_Win_lock_all, MPI_Win_unlock, MPI_Win_unlock_all` can be used to provide exclusive lock capability. These calls provide passive target communication and as mentioned earlier this kind of synchronization will not be supported.

5.3.1 Fence

This method of synchronization provides a global synchronization and it is performed using function:

```
int MPI_Win_fence(int assert, MPI_Win win)
```

This function in general starts and completes an access epoch and an exposure epoch. According to the specification [1] all operations will be completed at the origin before the fence call returns at the origin and they will be completed at the target by a matching call to `MPI_Win_fence` by the target process.

This call is collective on the group of `win` and as such in most cases (more information about it below) it may be used to provide barrier synchronization. This means that each call to `MPI_Win_fence` may be considered as a point of global consistency and data in persistent memory may be considered as a checkpoint, so this function will be modified to flush all the changes to persistent memory, commit the previous transaction and start a new one.

As noted above not always `MPI_Win_fence` has to act as a barrier. One example of this is when `MPI_Win_fence` does not end any epoch (e.g. when no communication call preceded it, see Section 11.5.1 of MPI specification [1]). To provide checkpointing we unfortunately need to imply that `MPI_Win_fence` always acts as a barrier. This however only applies when transactions are provided. When `pmem_dont_use_transactions` is set then the implementation will only flush persistent memory.

5.3.2 General active target synchronization

According to MPI specificaion [1] an MPI implementation must ensure that no one sided operations have access to a window before it is exposed by a call to `MPI_Win_post`. Synchronization between `MPI_Win_complete` and `MPI_Win_wait` ensures that all processes finish their communication before a window is unexposed. Weak synchronization means that operations on origin may end before window is exposed by target process if data is buffered by the implementation. According to the specification [1] all operations will have completed at the origin when `MPI_Win_complete` returns and they will be completed at the target by the matching call to `MPI_Win_wait` by the target process.

Since MPI offers only weak synchronization, origin processes do not have information whether their operations have finished on the target. This makes checkpointing a bit harder using this mechanism, because processes are not able to synchronize one with each other. However in master-slave applications if the master keeps all information about the application state then each call to `MPI_Win_wait` can be considered as making a consistent checkpoint.

General active target synchronization is performed by functions described below:

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
```

This function starts an access epoch. As described earlier only exposure epochs have impact on keeping data consistent, so this function does not need any modifications.

```
int MPI_Win_complete(MPI_Win win)
```

This function completes an access epoch. As described earlier only exposure epochs have impact on keeping data consistent, so this function does not need any modifications.

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
```

This function starts an exposure epoch. As described earlier this function needs to be modified to flush local changes to persistent memory, commit the previous transaction and start a new transaction.

```
int MPI_Win_wait(MPI_Win win)
```

This function completes an exposure epoch. As described earlier this function needs to be modified to flush local changes to persistent memory, commit the previous transaction and start a new transaction.

```
int MPI_Win_test(MPI_Win win, int *flag)
```

This function is a non-blocking version of `MPI_Win_wait`. This means that it needs identical modifications as `MPI_Win_wait`.

6 Other MPI methods

6.1 Initialization

There are two methods in MPI that initialize an MPI implementation:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Init_thread(int *argc, char ***argv, int required, int *provided)
```

Either of these must be called when an MPI application starts, so it is a good place to place initialization of error codes (see Section 4). Both of these functions will be modified to do so.

7 New methods for managing windows in persistent memory

This section provides information about new methods that need to be implemented. These methods are designed from scratch for the purpose of managing persistent memory windows used in one sided communication with checkpointing.

7.1 Setting root path

```
int MPI_Win_pmem_set_root_path(char *path)
```

This function has to be called before any window is created in persistent memory by `MPI_Win_allocate` or any window management function (see Section 7.2) is called. It saves a path that is used as a root path for all functions that search or allocate memory windows. The provided path must exist i.e. all directories on the path must exist.

7.2 Managing windows

This section provides information about methods used for listing available windows, listing available versions of windows and deleting versions and whole windows.

7.2.1 Listing windows information

```
int MPI_Win_pmem_list(MPI_Win_pmem_windows *windows)
```

This function gets information about all available windows in `MPI_Win_pmem_windows` opaque object. Information from this object may be retrieved by functions defined below.

```
int MPI_Win_pmem_free_windows_list(MPI_Win_pmem_windows *windows)
```

This function frees object allocated with `MPI_Win_pmem_list`.

```
int MPI_Win_pmem_get_nwindows(MPI_Win_pmem_windows windows, int *nwindows)
```

This function returns the number of available windows.

```
int MPI_Win_pmem_get_name(MPI_Win_pmem_windows windows, int n, char *name)
```

This function returns the name of the `n`th window in `windows`.

```
int MPI_Win_pmem_get_size(MPI_Win_pmem_windows windows, int n, int *size)
```

This function returns the size of the `n`th window in `windows`.

```
int MPI_Win_pmem_get_versions(MPI_Win_pmem_windows windows, int n,
    MPI_Win_pmem_versions *versions)
```

This function returns an opaque object with available versions of the `n`th window in `windows`.

```
int MPI_Win_pmem_free_versions_list(MPI_Win_pmem_versions *versions);
```

This function frees object allocated with `MPI_Win_pmem_get_versions`.

```
int MPI_Win_pmem_get_nversions(MPI_Win_pmem_versions versions,
    int *nversions)
```

This function returns the number of available versions in `versions`.

```
int MPI_Win_pmem_get_version(MPI_Win_pmem_versions versions, int n,
    int *version)
```

This function returns the `n`th available version in `versions`.

```
int MPI_Win_pmem_get_version_timestamp(MPI_Win_pmem_versions versions,
    int n, time_t *timestamp)
```

This function returns the `n`th available version's timestamp.

7.2.2 Deleting windows

```
int MPI_Win_pmem_delete(const char *name)
```

This function deletes a window with a given `name` (all of its versions and metadata).

```
int MPI_Win_pmem_delete_version(const char *name, int version)
```

This function deletes a specified `version` of a window with a given `name`.

8 Possible derivative works

This section gives an overview of possible derivative works that came up during design.

8.1 Application templates or frameworks

Since the proposed extensions impose some requirements for applications to apply checkpointing it is reasonable to provide templates how an application should be organized in order to benefit from the proposed extensions.

It is also possible to create a framework for automatic checkpointing of applications using the proposed extensions. In such a framework control flow will be predefined and the programmer will only have to implement missing parts of program logic. This way the programmer will not need to bother with checkpointing or even communication as it will be done automatically.

References

[1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*, September 2012.