Omni Compiler Runtime Library for Coarray

Akihiro Tabuchi

Ph.D. student at University of Tsukuba

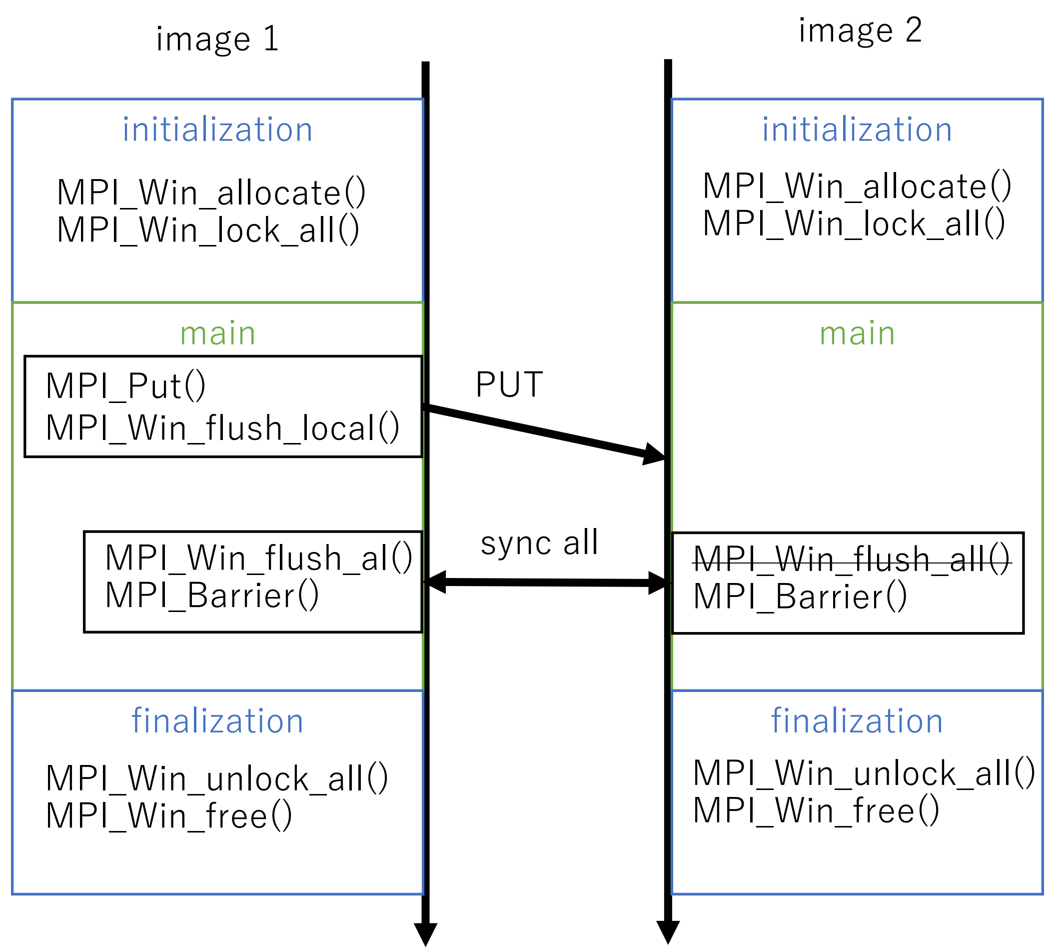
Omni compiler runtime library for coarray

- Supported one-sided communication library
  - MPI 3
  - GASNet
  - FJRDMA

Overview of implementation over MPI-3

- For the left program, the omni runtime library calls MPI functions as shown in the right figure.

```
int array[N]:[*];
void main()
{
  int buf[N] = ...;
  if (image_num == 1){
    array[:]:[2] = buf[:];
  }
  xmp_sync_all();
}
```



Initialization of coarray

xmp_init_all() calls _XMP_mpi_onesided_initialize() which setups MPI window

```
MPI_Win _xmp_onesided_win;  // window for coarray
void *_xmp_onesided_buf;    // base address of the window

void _XMP_mpi_onesided_initialize(int argc, char** argv, const size_t heap_size)
{
  MPI_Win_allocate(heap_size,             //window size
                   sizeof(char),          //gap size
                   MPI_INFO_NULL,
                   MPI_COMM_WORLD,
                   &_xmp_onesided_buf,
                   &_xmp_onesided_win);
  _XMP_mpi_build_shift_queue(false);      //init coarray heap
  MPI_Win_lock_all(0, _xmp_onesided_win);
}
```

- Parameter **heap_size** is coarray heap size. We can specify it by environment variable *XMP_ONESIDED_HEAP_SIZE*
- `MPI_Win_allocate()` creates a MPI window and allocates memory for coarray. All coarrays will belong to this window.
- `MPI_Win_lock_all()` begins an access epoch for the program. All coarray communication will be issued in this epoch.

Finalization of coarray

xmp_finalize_all() calls _XMP_mpi_onesided_finalize() which frees MPI window

```
void _XMP_mpi_onesided_finalize(void)
{
  MPI_Win_unlock_all(_xmp_onesided_win);
  _XMP_mpi_destroy_shift_queue(false);  // finalize coarray heap
  MPI_Win_free(&_xmp_onesided_win);
}
```

- `MPI_Win_unlock_all()` ends the access epoch began at _XMP_mpi_onesided_initialize().
- `MPI_Win_free()` frees the window and heap for coarray.

Allocation of coarray

_XMP_coarray_malloc()

```
void _XMP_coarray_malloc(void **coarray_desc, void **addr) {
  _XMP_coarray_t* c = _XMP_alloc(sizeof(_XMP_coarray_t));

  // set coarray info
  _XMP_coarray_set_info(c);
  *coarray_desc = c;
  size_t coarray_size = _total_coarray_elmts * _elmt_size;

  // allocate coarray from coarray heap
  _XMP_mpi_coarray_malloc(*coarray_desc, addr, coarray_size);

  // register coarray
  _push_coarray_queue(c);
}
```

- To create a coarray descriptor, `_XMP_coarray_set_info()` set coarray information specified by `_XMP_coarray_malloc_info_`*`X`*`()` and `_XMP_coarray_malloc_image_info_`*`X`*`()`
- `_XMP_mpi_coarray_malloc()` allocate the coarray from the coarray heap
- `_push_coarray_queue()` register the coarray descriptor.

PUT (continuous)

_XMP_coarray_continuous_put (special pattern, high-speed)

```
void _XMP_coarray_continuous_put(int target_image,
      void *dst_desc, void *src_desc, void *src_addr,
      long dst_offset, long src_offset, long dst_elmts, long src_elmts)
{
  int target_rank = target_image - 1;         //to 0-based
  if(target_rank == _XMP_world_rank){
    _XMP_local_continuous_copy(...); return;  //local memory copy
  }

  size_t transfer_size = dst_desc->elmt_size * dst_elmts;
  char *laddr = get_local_addr(src_desc, src_addr) + src_offset;
  char *raddr = get_remote_addr(dst_desc, target_rank) + dst_offset;

  MPI_Put(          laddr, transfer_size, MPI_BYTE, target_rank,
          (MPI_Aint)raddr, transfer_size, MPI_BYTE, _xmp_onesided_win);
  MPI_Win_flush_local(target_rank, _xmp_onesided_win);
}
```

- If `target_rank` is equal to my rank, it copies memory in local.
- `laddr` is local memory address and `raddr` is remote memory offset from the window base address.
- `MPI_Win_flush_local()` waits end of the PUT locally. (doesn't wait remote completion)

GET (continuous)

_XMP_coarray_continuous_get (special pattern, high-speed)

```
void _XMP_coarray_continuous_get(int target_image,
      void *dst_desc, void *dst_addr, void *src_desc,
      long dst_offset, long src_offset, long dst_elmts, long src_elmts)
{
  int target_rank = target_image - 1;         //to 0-based
  if(target_rank == _XMP_world_rank){
    _XMP_local_continuous_copy(...); return;  //local memory copy
  }

  size_t transfer_size = dst_desc->elmt_size * dst_elmts;
  char *laddr = get_local_addr(dst_desc, dst_addr) + dst_offset;
  char *raddr = get_remote_addr(src_desc, target_rank) + src_offset;

  MPI_Get(          laddr, transfer_size, MPI_BYTE, target_rank,
          (MPI_Aint)raddr, transfer_size, MPI_BYTE, _xmp_onesided_win);
  MPI_Win_flush_local(target_rank, _xmp_onesided_win);
}
```

- It is almost the same as `_XMP_coarray_continuous_put()`
- `MPI_Win_flush_local()` waits both local and remote completion of the GET

PUT (normal)

_XMP_coarray_put (general pattern)

```
void _XMP_coarray_put(void *dst_desc, void *src_addr, void *src_desc)
{
  int target_rank = calc_target_rank(_image_num, ...);
  if(target_rank == _XMP_wold_rank){
    _XMP_local_put(...); return; //local memory copy
  }

  char *laddr = calc_local_addr(src_addr, _src_info);
  char *raddr = calc_remote_addr(dst_desc, target_rank, _dst_info);
  MPI_Datatype src_types[_XMP_N_MAX_DIM], dst_types[_XMP_N_MAX_DIM];
  create_datatypes(src_types, _src_dims, _src_info, element_size);
  create_datatypes(dst_types, _dst_dims, _dst_info, element_size);
  MPI_Put(          laddr, 1, src_types[0], target_rank,
          (MPI_Aint)raddr, 1, dst_types[0], win);
  MPI_Win_Flush_local(target_rank, _xmp_onesided_win);
  free_datatypes(src_types, _src_dims);
  free_datatypes(dst_types, _dst_dims);
}
```

- `create_datatypes()` makes *N*-dimensional array datatype with `MPI_Type_create_hvector()`
  - `src_types[0]` is N-dim datatype, `src_types[1]` is (N-1)-dim datatype, …
  - Base datatype is `MPI_BYTE`

GET (normal)

_XMP_coarray_get (general pattern)

```
void _XMP_coarray_get(void *src_desc, void *dst_addr, void *dst_desc)
{
  int target_rank = calc_target_rank(_image_num, ...);
  if(target == _XMP_wold_rank){
    _XMP_local_put(...); return; //local memory copy
  }

  char *laddr = calc_local_addr(dst_addr, _dst_info);
  char *raddr = calc_remote_addr(src_desc, target_rank, _src_info);
  MPI_Datatype src_types[_XMP_N_MAX_DIM], dst_types[_XMP_N_MAX_DIM];
  create_datatypes(src_types, _src_dims, _src_info, element_size);
  create_datatypes(dst_types, _dst_dims, _dst_info, element_size);
  MPI_Get(         laddr, 1, src_types[0], target_rank,
          (MPI_Aint)raddr, 1, dst_types[0], win);
  MPI_Win_Flush_local(target_rank, _xmp_onesided_win);
  free_datatypes(src_types, _src_dims);
  free_datatypes(dst_types, _dst_dims);
}
```

- It is almost the same with `_XMP_coarray_put()`

Synchronization (1/2)

xmp_sync_all

```
void xmp_sync_all(int *status)
{
  if(!_is_win_flushed){
    MPI_Win_flush_all(_xmp_mpi_onesided_win);
    _is_win_flushed = true;
  }
  MPI_Barrier(MPI_COMM_WORLD);

  //this function returns no status now
}
```

- `_is_win_flushed` is false if some PUT/GETs are issued after last `MPI_Win_flush_all()`
- `MPI_Win_flush_all()` waits all end of preceding PUTs issued by this rank
- `MPI_Barrier()` synchronizes all images

Synchronization (2/2)

xmp_sync_images

```
void xmp_sync_images(int num, int *image_set, int *status)
{
  int rank_set[num];
  for(int i = 0; i < num; i++) rank_set[i] = image_set[i] - 1;

  if(!_is_win_flushed){
    MPI_Win_flush_all(_xmp_mpi_onesided_win);
    _is_win_flushed = true;
  }

  //send signals
  for(int i = 0; i < num; i++){
    MPI_Send(NULL, 0, MPI_BYTE, rank_set[i], TAG_SYNCREQ, COMM_WORLD);
  }

  //continue to the next slide
```

- `MPI_Win_flush_all()` waits end of all preceding PUTs
- `MPI_Send()` sends signals to ranks in `rank_set`

Synchronization (2/2, cont'd)

xmp_sync_images

```
  // cont'd
  // wait signals
  for(int rank : rank_set){
    if( _signal_table[rank] > 0 ){
      _signal_table[rank] -= 1;
      remove(rank_set, rank);
    }
  }
  while( size(rank_set) > 0 ){
    MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE, TAG_SYNCREQ, MPI_COMM_WORLD, &st);
    int rank = st.MPI_SOURCE;
    if( contains(rank_set, rank) )
      remove(rank_set, rank);
    else
      _signal_table[rank] += 1;
  }

  //this function returns no status now
}
```

`_signal_table[]` has # of received signals

- Firstly, matching with already received signals in `_signal_table`
- `MPI_Recv()` receives a signal from any ranks. If `rank_set` contains the rank, it is removed from `rank_set`, else it is added to `_signal_table`. This is repeated while `rank_set` is not empty.

Question

- MPI 3.0 spec. p.459, Example 11.9: the following example is unsafe even in the unified model, because the load of X can not be guaranteed to occur after the MPI_BARRIER. ... Compiler and hardware specific notations could ensure the load occurs after the data is updated, or explicit one-sided synchronization calls can be used to ensure the proper result.

```
Process A:                                  Process B:
                                            window location X
MPI_Win_lock_all
MPI_Put(X) /* update to window */
MPI_Win_flush(B)
MPI_Barrier                                 MPI_Barrier
                                            load X


MPI_Win_unlock_all
```

- We have to insert one-sided synchronization call to ensure result. Is the following code correct?

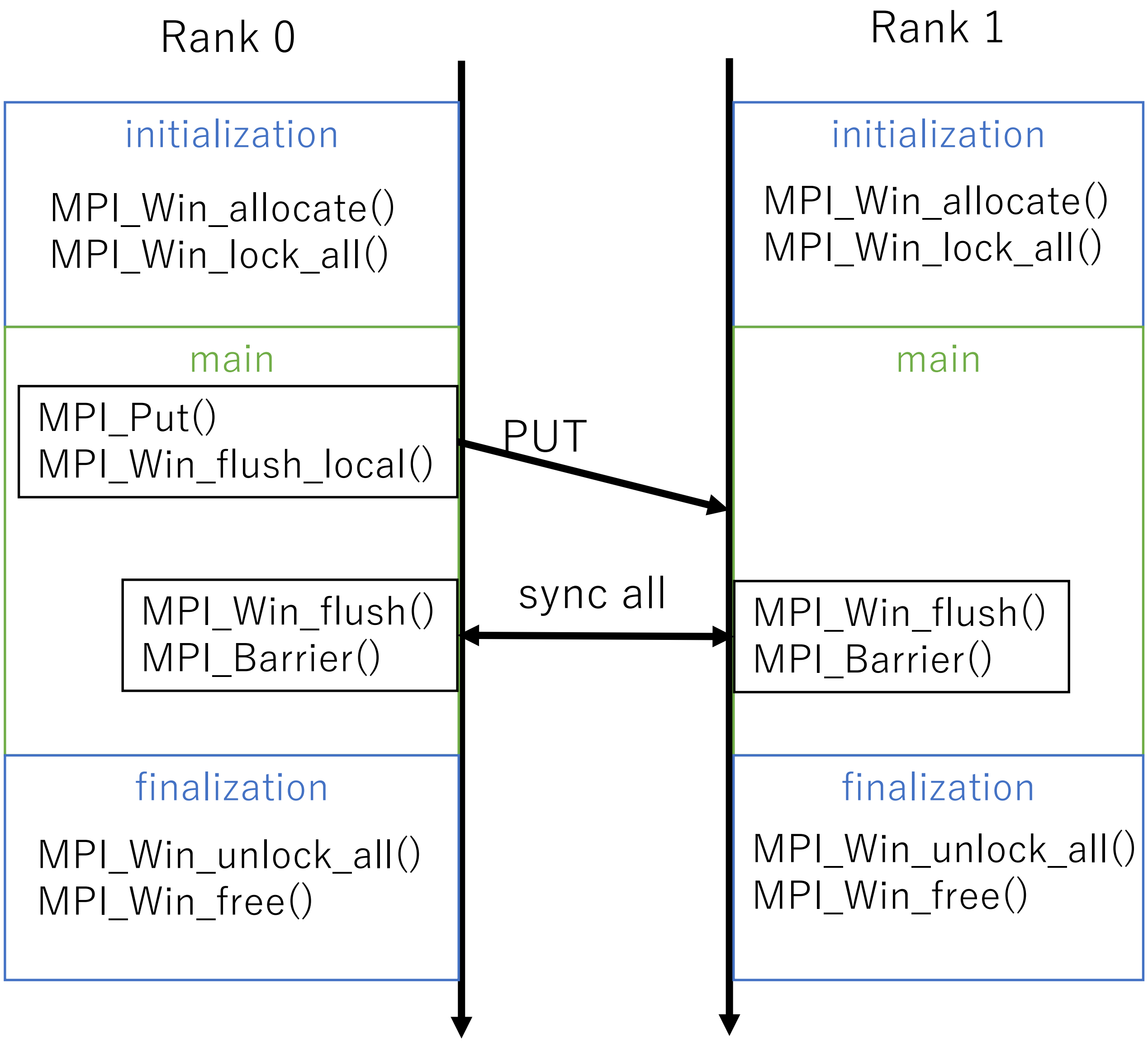
```
Process A:                                  Process B:
                                            window location X
MPI_Win_lock_all
MPI_Put(X) /* update to window */
MPI_Win_flush(B)
MPI_Barrier                                 MPI_Barrier
                                            MPI_Win_sync
                                            load X

MPI_Win_unlock_all
```

- and, should we change synchronization functions like the following?

```
void xmp_sync_all(int *status)
{
  if(!_is_win_flushed){
    MPI_Win_flush_all(_xmp_mpi_onesided_win);
    _is_win_flushed = true;
  }
  MPI_Barrier(MPI_COMM_WORLD);
}
```

```
void xmp_sync_all(int *status)
{
  if(!_is_win_flushed){
    MPI_Win_flush_all(_xmp_mpi_onesided_win);
    _is_win_flushed = true;
  }
  MPI_Win_sync(_xmp_mpi_onesided_win);
  MPI_Barrier(MPI_COMM_WORLD);
}
```


Rank 0
Rank 1
initialization
MPI_Win_allocate()
MPI_Win_lock_all()
initialization
MPI_Win_allocate()
MPI_Win_lock_all()
main
MPI_Put()
MPI_Win_flush_local()
PUT
main
MPI_Win_flush()
MPI_Barrier()
sync all
MPI_Win_flush()
MPI_Barrier()
finalization
MPI_Win_unlock_all()
MPI_Win_free()
finalization
MPI_Win_unlock_all()
MPI_Win_free()