HLRN Parallel Programming Workshop

Speedup your Code on Intel® Processors at HLRN

October 20th, 2017

MPI-3 One-Sided Communication

Florian Wende

Zuse Institute Berlin
Two-sided communication

Standard Message Passing Interface (MPI) draws on a two-sided communication and collective communication model

- sender and receiver participate in data exchange explicitly
  → synchronization: sending process has to wait for the receiver to start receiving
- memory is private to each process
Two-sided communication

Standard Message Passing Interface (MPI) draws on a two-sided communication and collective communication model

- sender and receiver participate in data exchange explicitly
  - synchronization: sending process has to wait for the receiver to start receiving
- memory is private to each process
Two-sided communication

Standard Message Passing Interface (MPI) draws on a two-sided communication and collective communication model

- sender and receiver participate in data exchange explicitly
  → synchronization: sending process has to wait for the receiver to start receiving
- memory is private to each process

What about `MPI_Isend` and `MPI_Irecv` and all these “non-blocking” versions of the MPI communication primitives?

- kind of ticket system to decouple the intent to send/receive data from actually doing so
  → get a ticket (request) you can later synchronize on
One-sided communication

MPI-2 already introduced one-sided communication through remote memory access (RMA)

- only one process is required for the data transfer
- decoupling of data transfer and process synchronization
  → multiple data transfers with a single synchronization operation
- increased performance on systems with RDMA support
One-sided communication

MPI-2 already introduced one-sided communication through remote memory access (RMA)
- only one process is required for the data transfer
- decoupling of data transfer and process synchronization
  → multiple data transfers with a single synchronization operation
- increased performance on systems with RDMA support

Diagram:
- Two-sided communication:
  - Two arrows between process 0 and process 1 labeled "comm + sync".
- One-sided communication:
  - Arrow from process 0 to process 1 labeled "comm".
  - Arrow from process 1 to process 0 labeled "sync".
MPI-3 – One-sided communication

Revision and extension of MPI-2 RMA to improve the performance

- `MPI_Get`
- `MPI_Put`
- `MPI_Accumulate`
- `MPI_Win_create`
- `MPI_Win_free`
- `MPI_Win_fence`

... means to access RDMA capabilities of the hardware if present

RMA = programming interface
RDMA = hardware implementation of RMA

No guarantee that communication actually happened before synchronizing all participating processes, e.g. via `MPI_Win_fence`!
MPI-3 – Memory model

*Local memory*
any memory allocation is by default only locally accessible

*Remotely accessible memory*
region of the local memory explicitly made accessible to all other processes
Local memory
any memory allocation is by default only locally accessible

Remotely accessible memory
region of the local memory explicitly made accessible to all other processes
Creating remotely accessible memory (*collective operation*)

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

- **base**: pointer to local memory
- **size**: size of the local memory to be remotely accessible
- **disp_unit**: local unit size for displacement
- **info**: info argument (handle)
- **comm**: communicator (handle)
- **win**: window object (handle)

![Diagram showing window objects on processes 0, 1, 2, and 3]
Creating remotely accessibly memory (*collective operation*)

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

```c
MPI_Win_create_dynamic(MPI_info info, MPI_Comm comm, MPI_Win* win)
MPI_Win_attach(MPI_Win win, void* base, MPI_Aint size)
```

*create a dynamic window and attach memory to it later*
Creating remotely accessibly memory (collective operation)

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

MPI_Win_create_dynamic(MPI_info info, MPI_Comm comm, MPI_Win* win)

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

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

MPI will allocate local memory and returns a pointer to it
Creating remotely accessibly memory (*collective operation*)

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

```c
MPI_Win_create_dynamic(MPI_info info, MPI_Comm comm, MPI_Win* win)
MPI_Win_attach(MPI_Win win, void* base, MPI_Aint size)
```

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

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

*MPI will allocate local memory and returns a pointer to it: the allocated memory is contiguous across all processes in shared memory (the user has to make sure that all processes in comm can create a shared memory segment)*
MPI-3 – Memory model (RMA windows)

Creating remotely accessibly memory (*collective operation*)

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

MPI_Win_create_dynamic(MPI_info info, MPI_Comm comm, MPI_Win* win)
MPI_Win_attach(MPI_Win win, void* base, MPI_Aint size)

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

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

MPI_Win_free(MPI_Win* win)
```
Once created a window object, data can be moved through it by any remote process, thereby altering the content of the underlying local memory.

```c
MPI_Put(void* origin, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
```
Once created a window object, data can be moved through it by any remote process, thereby altering the content of the underlying local memory.

```c
MPI_Get(void* origin, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target Disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)
```
Example

```c
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int proc_id, num_procs;
    MPI_Comm_rank(MPI_COMM_WORLD, &proc_id);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    int* local_mem = (int*) malloc(N * sizeof(int));
    int* shared_mem = (int*) malloc(N * sizeof(int));

    for (int i = 0; i < N; ++i) {
        local_mem[i] = proc_id * N + i;
        shared_mem[i] = 0;
    }
    ...
```
MPI-3 – Data transfer I

Example (continued)

..

MPI_Win win;
MPI_Win_create(shared_mem, N, sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &win);

MPI_Win_fence(0, win);
MPI_Put(local_mem, N, M_INT, (proc_id + 1) % num_procs, 0, N, MPI_INT, win);
MPI_Win_fence(0, win);

cout << "mpi rank " << proc_id << " got data: ";
for (int i = 0; i < (N - 1); ++i)
    cout << shared_mem[i] << ",";
cout << shared_mem[N - 1] << endl;

MPI_Win_free(&win);
MPI_Finalize();
return 0;
}
Example (continued): build and run on the Cray

```bash
$> module load dmapp
$> export MPICH_RMA_OVER_DMAPP=1
$> export MPICH_RMA_USE_NETWORK_AMO=1

$> CC -O2 -Wl,--whole-archive,-ldmapp,--no-whole-archive -o mpi3_put.x mpi3_put.cpp
```
Example (continued): build and run on the Cray

```bash
$> module load dmapp
$> export MPICH_RMA_OVER_DMAPP=1
$> export MPICH_RMA_USE_NETWORK_AMO=1

$> CC -O2 -Wl,--whole-archive,-ldmapp,--no-whole-archive -o mpi3_put.x mpi3_put.cpp

$> aprun -n 2 ./mpi3_put.x
```

```
0 1 2 3 | 0 0 0 0
P0

4 5 6 7 | 0 0 0 0
P1
```
Example (continued): build and run on the Cray

```
$> module load dmapp
$> export MPICH_RMA_OVER_DMAPP=1
$> export MPICH_RMA_USE_NETWORK_AMO=1

$> CC -O2 -Wl,--whole-archive,-ldmapp,--no-whole-archive -o mpi3_put.x mpi3_put.cpp

$> aprun -n 2 ./mpi3_put.x
```

![Diagram of MPI_Put(..) + MPI_Win_fence(..)]
MPI-3 – Synchronization models

When is a process allowed to read/write remotely accessible memory?

- fence (active target)
- post-start-complete-wait (generalized active target)
- lock/unlock (passive target)
When is a process allowed to read/write remotely accessible memory?

**fence** (active target)

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

- collective synchronization: all processes in the group of `win` participate
- time period between two fences defines an “epoch”
- every process can issue an arbitrary number of puts/gets within an epoch
When is a process allowed to read/write remotely accessibly memory?

**post-start-complete-wait** (generalized active target)

posting processes

```c
MPI_Win_post(MPI_Group group, int assert, MPI_Win win_group)
MPI_Win_wait(win_group)
```

starting processes

```c
MPI_Win_start(MPI_Group group, int assert, MPI_Win win_group)
MPI_Win_complete(win_group)
```

- posting processes start an “exposure epoch”
- starting process then can safely accessing the data of the posting processes
- general: $n$ posting and $m$ starting processes
MPI-3 – Synchronization models

When is a process allowed to read/write remotely accessible memory?

**lock/unlock** (passive target)

\[
\text{MPI\_Win\_lock}(\text{int lock\_type, int target\_rank, int assert, MPI\_Win win})
\]

\[
\text{MPI\_Win\_unlock}(\text{int target\_rank, MPI\_Win win})
\]

lock\_type = MPI\_LOCK\_EXCLUSIVE | MPI\_LOCK\_SHARED

- calling process locks the remotely accessible memory of target\_rank associated with win
- MPI\_LOCK\_SHARED basically corresponds to an “exposure epoch” without the need of the target\_rank to actively participate that process
Elementwise atomic update of data in an RMA window

```c
MPI_Accumulate(void* origin, int origin_count, MPI_Datatype origin_datatype,
    int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
    MPI_Op op, MPI_Win win);
```

- `op` = `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, ..., `MPI_REPLACE`
- only pre-defined reduce operations
- reduction happens on the target data
- `MPI_REPLACE`: atomic “put” operation
Elementwise atomic read-update of data in an RMA window

```c
MPI_Get_accumulate(void* origin, int origin_count, MPI_Datatype origin_datatype,
void* result, int result_count, MPI_Datatype result_datatype,
int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype,
MPI_Op op, MPI_Win win);
```

- `op = MPI_MAX, MPI_MIN, MPI_SUM, ..., MPI_REPLACE`
- only pre-defined reduce operations
- reduction happens on the target data
- `MPI_REPLACE`: atomic put operation
Elementwise atomic read-update of data in an RMA window

\[
\text{MPI\_Fetch\_and\_op}(\text{void* origin, void* result, MPI\_Datatype datatype, int target\_rank, MPI\_Aint target\_disp, MPI\_Op op, MPI\_Win, win})
\]

\[
\text{MPI\_Compare\_and\_swap}(\text{void* origin, void* compare, void* result, MPI\_Datatype datatype, int target\_rank, MPI\_Aint target\_disp, MPI\_Win, win})
\]