One sided communication

George Bosilca

Remote Memory Access

- Cooperative way of moving data around (the point-to-point layer) involve both the sender and the receiver in the operation
- Changes in the memory of the receiver can happen only when the receiver allows them
- Changes happen only at the specified memory addresses
- MPI point-to-point semantics force the ordering of the messages between 2 peers
- The cost of the message matching is considered overwhelming
- Both the sender and the receiver are involved in the communication, they both have to call a specific MPI function
- Performance degradation
- Limit the expressiveness of the application as every send have to match a receive.
- Difficult to create programs where the number of messages 2 peers have to exchanges is not similar
- Example: a shared counter

Remote Memory Access

- Lead to a simpler view of the program: sends and receives match, the communications are clearly expressed by the semantic of the application
- Therefore, RMA is here to save us!
- Only one function required to perform the data movement (which specify the source as well as the destination)
- Performance improvement but not necessary for ping-pong benchmark (think overlapping communication with computation)
- No matching!!! No ordering!!!

One sided vs. Point-to-point

If( 0 == rank ) {
  MPI_Isend( to_1 );
} else if( 1 == rank ) {
  MPI_Irecv( from_0 );
}
MPI_Wait()

If( 0 == rank ) {
  MPI_Win_create( NULL, 0,
  MPI_INFO_NULL, comm,
  &win );
} else if( 1 == rank ) {
  MPI_Win_create( addr,
  length, sizeof(int),
  MPI_INFO_NULL, comm,
  &win );
}
MPI_Win_fence( 0, win );

If( 0 == rank ) {
  MPI_Put( to_remote_mem )
  MPI_Win_fence( 0, win );
}
MPI_Win_free( &win )

Memory Windows

- Define a contiguous memory region that will be available to peers as a target for their put/get or accumulate operations, described by the base address and the length in bytes.
- It is a local memory! Each process can specify a different base and a different length, but they all have to specify the same communicator.
- It define not only the memory available but the processes who have the right to modify it.
- Both MPI_Win_create and MPI_Win_free are collective calls over the involved processes (the communicator).

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

int MPI_Win_free( MPI_Win* win )

Memory Windows

- Special arguments:
  - disp_unit
    - It’s a basic displacement unit for the remote displacement
    - When do we need it?
    - Why not using always byte displacement?
  - Info
    - Giving hints to the MPI library to allow performance improvement
    - It’s a set of guarantee that the user provide to the MPI library
Memory Windows

- Options for MPI_Win_fence
  - MPI_MODE_NOSTORE: local windows was not updated by local stores since the last call to MPI_Win_fence.
  - MPI_MODE_NOPUT: the local window will not be updated by put or accumulate between this fence and the next fence call
  - MPI_MODE_NOPRECEDE: the fence will not complete any RMA calls made by the process calling MPI_Win_fence. It should be global.
  - MPI_MODE_NOSUCCEED: No RMA call will be made on this windows between this and the next call to fence.

Moving data

```c
int MPI_Put( void* origin_addr, int origin_count, MPI_Datatype origin_datatype,
            int target_rank, MPI_Aint target_disp, int target_count,
            MPI_Datatype target_datatype, MPI_Win win )
```

*same prototype for MPI_Get*

Moving data

Completing data transfers

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

Example

- Poisson resolution: the ghost cell mesh
Example

- Poisson resolution: the ghost cell mesh

Example

- Exchange the data using send/recv

Example

- Exchange the data using one sided (PUT)

Example

- Exchange the data using one sided (PUT & GET)

Example

- Now imagine a two-dimensional mesh and exchange the data using one-sided
Accumulate

- Apply a specific operation on remote memory.
- The operation should be a predefined one.
- The data-type should be predefined as well.
- Concurrent updates using accumulate
  - Special operation: MPI_REPLACE

MPI_Accumulate(void* origin_addr, int origin_count, MPI_Datatype orig_ddt, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_ddt, MPI_Op op, MPI_Win win)

Lock and Unlock

- MPI_Win_fence is an active target synchronization.
- The passive target synchronization is when the target process is not required to call any MPI functions.
- Same operations can be used: put, get, accumulate except they should be surrounded by locks.
- Type ca be:
  - MPI_LOCK_SHARED: several RMA operations are allowed to act on the same window
  - MPI_LOCK_EXCLUSIVE: request exclusive lock to the remote window.

int MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)
int MPI_Win_unlock(int rank, MPI_Win win)

Scalable synchronization

- A lighter version of the MPI_Win_fence
- Exposure epoch: the period of time when a local window may be target of RMA operations.
- MPI_Win_post begin the exposure epoch and MPI_Win_wait ends it.
- Post and wait open the local window to remote processes in the from_group
- Start/complete marks the processes in the to_group as target for one sided communication in the specified epoch.

int MPI_Win_start(MPI_Group to_group, int assert, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
int MPI_Win_post(MPI_Group from_group, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)

Scalable synchronization

- Assert can be:
  - MPI_MODE_NOSTORE: no local stores will affect the local window since the last call to MPI_Win_complete
  - MPI_MODE_NOPUT: the local window will not be updated by put or accumulate calls between this call to post and the corresponding call to complete.
  - MPI_MODE_NOCHECK: the matching MPI_Win_start calls have not been issued by any process that is an origin of RMA operations that have this process as target. The corresponding MPI_Win_start has to specify the same assert.

Example

- Use the same example as before but now use post and start.

Homework

Distributed Last In First Out (LIFO) lists.

- A LIFO is a stack having the following property:
  - The last element inserted will be the first element extracted.
- Therefore, there are only 2 operations on a lifo:
  - POP: extracting the last element
  - PUSH: adding a new element.
Homework

- The homework consists of implementing a distributed LIFO over one-sided operations in MPI.
- You have to implement 4 functions as described in the next paragraphs. There are 2 types of functions: collective calls used to create and destroy the stack, and manipulation functions used to add and remove an element to/from the distributed stack.

Initialization/destruction functions:
1. `distributed_stack_t* stack_init( MPI_Comm, int stack_size )`
   This is a collective function called by all processes on the communicator. It will initialize a distributed stack containing at most stack_size number of integers. It will return your own distributed_stack_t* pointer to the newly created stack or NULL if some errors have been detected.

2. `int stack_destroy( distributed_stack_t* )`
   This is a collective function called by all processes involved in the communicator that has been used to create the distributed stack. It has to release all resources used by the stack. It return MPI_SUCCESS if there were no errors and MPI_ERROR_OTHER otherwise.

Manipulation functions:
1. `int stack_push( distributed_stack_t* stack, int value )`
   Atomically add the value into head of the stack. It return MPI_SUCCESS if the element was correctly inserted, MPI_ERR_ARG if there is no more space on the stack and MPI_ERR_OTHER otherwise.

2. `int stack_pop( distributed_stack_t* stack, int* returned_value )`
   Atomically remove a value from the head of the stack. If the return is MPI_SUCCESS then the returned_value contain the integer removed from the distributed stack. Otherwise, the returned_value is not significant. If the stack is empty the function will return MPI_ERR_ARG, otherwise it will return MPI_ERR_OTHER.

There are 3 things that have to be taken in account (the number gives their priority):
1. Correctness: if all push succeed, then there is no missing element.
2. Performance:
3. Fairness: all the processes involved have the same priority regarding the 2 manipulation functions (push and pop).

Questions ...