# MPI3-RMA: A flexible, high-performance RMA interface for MPI

September 4, 2008

## 1 Introduction

.

MPI-2 standard defined RMA interfaces; the standard states:

> The design of the RMA functions allows implementors to take advantage, in many cases, of fast communication mechanisms provided by various platforms, such as coherent or noncoherent shared memory, DMA engines, hardware-supported put/get operations, communication coprocessors, etc.

In practice this portion of the standard has not been embraced by the user community because of the lack of high-performance implementation of this interface. In addition, this interface exposes in detail memory coherency issues, rendering the use of these interfaces more complex than users are willing to deal with.

The notion of RMA communications has the promise of providing high-performance communications, mapping well onto modern network stacks, and ease of use for non predictable communication patterns. Therefore, there is good reason to reconsider the RMA interface supported by the MPI standard, with an eye to making this interface attractive to many application developers. The intent to provide a portable interface that works correctly on the full range of computer platforms, minimizing the effort required by application developers to manage memory coherency. Specifically, the goal is to allow the interface users to specify a set of attributes the one-sided communications should satisfy, and leave it to the implementation to manage these communications in the way suitable for the given platforms, with out exposing to these users abstractions such as memory windows. These attributes include item such as cache coherency management and RMA ordering semantics.

## 2 Related RMA models

Several libraries support RMA interfaces that are internally used by many GAS languages and libraries. We have evaluated the semantics of SHMEM [1] ARMCI [2] GASNet [3] RMA interfaces in coming up with our design of MPI3-RMA interfaces.

## 3 A different approach

Our approach is based on the both our analysis of different RMA models and our understanding of how users use RMA. Many users perceive RMA models as extensions of load/store type access to a distributed environment; RMA naturally enables a partitioned global address space model.

However, the RMA communication in itself has certain attributes that are effected by both the underlying network and the machine/memory model. Hiding characteristics of a network or the machine model from the programmers in the RMA interfaces they use has two significant disadvantages

1. Users do not think of *which* of the attributes their particular usage of the RMA model warrants

2. Users go into misguided perception about cost of supporting attributes that are not inherent in some architectures but are supported on others ( eg. Ordering or messages is something the one-sided model I use guarantees, hence it should ubiquitously not bear any additional overhead)

Our approach is different in that the RMA interfaces, by default, offer no additional guarantees other than what the underlying machine and the network already support; any additional requirements are met by explicitly configuring attributes and there by being aware and preparing for potential impact on performance.

Our approach enables the user to program for performance.

## 4 Terminology

Through this document, we discuss the following different scenarios:

1. Two different MPI tasks communicating to the same destination to overlapping memory regions with any combination of the supported Put, Get and Accumulate operations

2. One MPI task doing simultaneous RMA operations to the same destination MPI task

There are several terms used within this document. Our usage of these terms is with the following definitions in mind:

1. RMA: Remote Memory Access

2. LMA: Local Memory Access, either directly through a pointer like mechanism or through a system call

3. Origin, Target: MPI tasks that are involved in RMA. Similar to their definition in MPI2 for a put operation, source=origin and destination=target; in a get operation, source=target and destination=origin.

4. Origin_addr: The initial address of buffer at the origin of the data transfer

5. Target_addr: The initial address of buffer at the target of the data transfer.

6. Local Completion: An RMA operation is complete at the Origin for Put/Accumulate and at the Target for Get.

7. Remote Completion: Applies to Put and Accumulate, the RMA operation is complete at the Target and the Target_addr reflects the result of the operation

8. Consistency: All references to Consistency are at the granularity of an RMA operation

9. Atomicity Guarantee: A single Atomic RMA operation will execute with a guarantee that no other Atomic RMA operation will update the contents at either its Origin_addr or on its Target_addr between its start and finish

## 5 Flexible, high-performance RMA interfaces

### 5.1 Interfaces for communication

In order to allow for flexibility and to achieve performance, we propose the following communication calls: **MPI_RMA_xfer**, **MPI_RMA_rmw**, and **MPI_RMA_rmw2**.

MPI_RMA_xfer has the following Operation Types: GET, PUT, ACCUMULATE. Similar to the definitions in the MPI2 standard for RMA interfaces, GET transfers from the caller memory (origin) to the target memory; PUT transfers data from the target memory to the caller memory; and ACCUMULATE updates locations in the target memory. These RMA operations are non-blocking by default and allow for some configurable attributes.

MPI_RMA_rmw has the following Operations Types:

| | |
|---|---|
| MPI_RMW_INC | increment |
| MPI_RMW_PROD | product |
| MPI_RMW_SUM | sum |
| MPI_RMW_LAND | logical and |
| MPI_RMW_LOR | logical or |
| MPI_RMW_LXOR | logical xor |
| MPI_RMW_BAND | binary and |
| MPI_RMW_BOR | binary or |
| MPI_RMW_BXOR | binary xor |
| MPI_RMW_SWAP | swap value |

The target address is updated according to the specified operation using the value at operand_addr as the operand to the operation. result_addr is updated with the value at the target address prior to the update. Concurrent RMW operations specifying the same (or overlapping) target address are allowed and the updates occur as if the operations occurred in some order. By default the operation is non-blocking with the same completion semantics as other RMA operations. Note that operand_addr and target must be the same datatype and count

MPI_RMA_rmw2 has the following Operations Types:

| | |
|---|---|
| MPI_RMW2_MASK_SWAP | swap masked bits |
| MPI_RMW2_COMP_SWAP_LT | compare and swap (<) |
| MPI_RMW2_COMP_SWAP_LE | compare and swap (<=) |
| MPI_RMW2_COMP_SWAP_E | compare and swap (==) |
| MPI_RMW2_COMP_SWAP_GE | compare and swap (>=) |
| MPI_RMW2_COMP_SWAP_GT | compare and swap (>) |
| MPI_RMW2_COMP_SWAP_NE | compare and swap (! =) |

MPI_RMW2_COMP_SWAP: Compares the comperand (operand_addr) with the target value if comperand is $(<, <=, >, >=, ==, ! =)$ then the target value is updated with the swaperand (operand2_addr) value. Result address is updated with the target value prior to the compare/swap.

MPI_RMW2_MASK_SWAP: Updates the values of target specified in the maskerand (operand_addr) with the corresponding values in the swaperand (operand2_addr). Result address is updated with the target value prior to the mask/swap.

#### 5.1.1 Attributes of an RMA operation

MPI_RMA_xfer communication can have the following configurable attributes: Atomicity, Ordering, blocking and remote completion. **When none of these attributes are set, which is the default case, the RMA operation has no atomicity guarantee, ordering is not ensured, calls**

**are non-blocking and completion of a call only satisfies local completion requirements**. In the future other attributes like *Fault Tolerance guarantee* may be easily added to this list. The attributes are specifically designed to provide the programmers with a choice of guarantees they can expect from a subset of their RMA operation. **These attributes may be set per call or per communicator**. An attribute is applied to a call as long as it is set either at the call level or at the communicator level.

**Atomicity attribute** Any RMA operation with Atomicity attribute will complete execution without overlapping update to Target or Origin by any other RMA operation with the same attribute. Two operations with Atomicity attribute are not guaranteed to execute in order.

**Ordering attribute** By default, no particular ordering of RMA operations is guaranteed. Any two RMA operations with Ordering attribute and the same Target and overlapping Target_addr are assured to execute in order with respect to each other. A GET operation with the same Origin as the Target of a previous Put and overlapping Origin_addr and Target_addr with this attribute will execute in order with respect to the PUT operation. The objective of Ordering is oblivious to sequential consistency; it is to guarantee write consistency.

**Blocking attribute** All RMA operations are non-blocking in nature. They can be completed with explicit completion calls. By setting the Blocking attribute, the RMA operation blocks on completion.

**Remote Completion attribute** By default, completion only applies to the local side of the data transfer. This means, completion of a PUT or an ACCUMULATE operation is the assurance to the programmer that the Origin_addr buffer may be safely reused. By setting the remote completion attribute, a completion of an RMA operation will now indicate remote completion. This applies to PUT and ACCUMULATE operations. When the Remote Completion attribute is set for PUT or ACCUMULATE, completion guarantees that the operation has been completed on the Target and the data has been committed to memory.

Another additional attribute is discussed in 5.4

### 5.1.2 Semantics and Correctness

For overlapping update Origin_addr and Target_addr via RMA or a combination of RMA and LMA, the end result of the operation is not defined. Sequential consistency is not guaranteed (it however can be ensured by following correct semantics and setting the appropriate attributes). The RMA will use general datatype arguments to specify communication buffers at the origin and at the target. For all three Operation Types of RMA_xfer, the target process may be identical with the origin process; i.e., a process may use an RMA operation to move data in its memory. Hence, self-communication is supported.

Although nothing prevents it, a safe program should follow the following rules:

1. The Origin_addr or Target_addr in an RMA call should not be updated (via LMA or other RMA) after the call begins and before completion.

2. While an RMA transfer with Atomicity attribute set is in progress, memory at Origin_addr and Target_addr may not be accessed by other RMA operations that do not have the Atomicity attribute set

## 5.2 Interfaces for Remote Completion and Ordering

Both remote completion and ordering can be specified on a per request basis using the attributes detailed above. In addition to per-request attributes remote completion and ordering can be achieved by using the **MPI_RMA_Complete** and **MPI_RMA_Fence**. Both function calls are local to the process requesting remote completion or ordering of requests (these are not collective calls). MPI_RMA_Complete is used by a process to request remote completion of all outstanding RMA requests to a specified rank or an entire communicator if MPI_ALL_RANKS is specified. When MPI_RMA_Complete returns, all previously issued RMA operations are completed at the specified targets. MPI_RMA_Fence is used by a process to request ordering of a subsequent RMA operations with respect to all previously issued RMA operations to the specified targets. All RMA operations issued before a call to MPI_RMA_Fence are guaranteed to be visible remotely before any RMA operations issued after the call to MPI_RMA_Fence for the ranks/communicator specified in the MPI_RMA_Fence call.

### 5.2.1 Semantics and Correctness

Data accessed by RMW operations may not be accessed by any form of LMA. This violates any guarantees that the operation provides.

## 5.3 Example interfaces

The prototypes of the MPI_RMA_xfer functions are shown below. The first prototype takes in attributes of an individual call as a parameter. The last two prototypes propose two new functions, MPI_Req_set_attr and MPI_Req_get_attr to set and get attributes. If MPI_Request is used as the request data structure, these attributes will be per MPI_Request. Alternatively a new request structure MPI_Rma_request may also be used as shown in the third prototype.

```
//if MPI_Request is used in MPI_RMA_xfer and attributes are passed
//as a parameter
MPI_RMA_xfer( MPI_Op RMA_OP_TYPE, void *origin_addr, int origin_count,
              MPI_Datatype origin_datatype, int target_rank,
              void *target_addr, int target_count,
              MPI_Datatype target_datatype, MPI_Request *request,
              MPI_Comm *communicator,void *RMA_ATTR_VAL);

//if MPI_Request is used in MPI_RMA_xfer
MPI_RMA_xfer( MPI_Op RMA_OP_TYPE, void *origin_addr, int origin_count,
              MPI_Datatype origin_datatype, int target_rank,
              void *target_addr, int target_count,
              MPI_Datatype target_datatype, MPI_Request *request,
              MPI_Comm *communicator);

MPI_Req_set_attr(MPI_Request *req, int comm_keyval,
                 void *attribute_val);
MPI_Req_get_attr(MPI_Request *req, int comm_keyval,
                 void *attribute_val, int *flag);

//if a new MPI_RMA_Request is used in MPI_RMA_xfer
```

```
MPI_RMA_xfer( MPI_Op RMA_OP_TYPE, void *origin_addr , int origin_count ,
              MPI_Datatype origin_datatype , int target_rank ,
              void *target_addr , int target_count ,
              MPI_Datatype target_datatype , MPI_Rma_request *request ,
              MPI_Comm *communicator );
MPI_Req_set_attr(MPI_Rma_request *req , int comm_keyval,
                 void *attribute_val);
MPI_Req_get_attr(MPI_Rma_request *req , int comm_keyval,
                 void *attribute_val , int *flag );
```

Errors: MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_RANK

The prototypes of the MPI_RMA_rmw and MPI_RMA_rmw2 functions are shown below. The first prototype takes in attributes of an individual call as a parameter. The last two prototypes propose two new functions, MPI_Req_set_attr and MPI_Req_get_attr to set and get attributes. If MPI_Request is used as the request data structure, these attributes will be per MPI_Request. Alternatively a new request structure MPI_Rmw_request may also be used as shown in the third prototype.

```
MPI_RMA_rmw(MPI_RMW_Op op , void *operand_addr , int count ,
            MPI_Datatype datatype , void *result_addr ,
            int target_rank , MPI_Aint target_disp ,
            MPI_Rmw_request *request ,
            MPI_Comm *communicator , void *RMW_ATTR_VAL)


MPI_RMA_rmw2(MPI_RMW_2_Op op , void *operand_addr , int count ,
             MPI_Datatype datatype , void *operand2_addr ,
             void *result_addr , int target_rank ,
             MPI_Aint target_disp , MPI_Rmw_request *request ,
             MPI_Comm *communicator , void *RMW_ATTR_VAL)

MPI_Req_set_attr(MPI_Rmw_request *req , int comm_keyval,
                 void *attribute_val);
MPI_Req_get_attr(MPI_Rmw_request *req , int comm_keyval,
                 void *attribute_val , int *flag );
```

Errors: MPI_SUCCESS, MPI_ERR_COMM, MPI_ERR_COUNT, MPI_ERR_TYPE, MPI_ERR_RANK

The prototypes of the MPI_RMA_Complete and MPI_RMA_Fence functions are shown below.

```
MPI_RMA_Complete(MPI_Comm *communicator , int target_rank );

MPI_RMA_Fence(MPI_Comm *communicator , int target_rank );
```

### 5.4 Interfaces for Memory allocation

Memory allocation interfaces are required for two reasons: a) allocation interface provides explicit mechanism to get information about remote memory, either a particular region or some accessible section of the memory b) allocation interfaces provide mechanisms to allocate communicatable memory in situations where direct virtual memory communication requires additional steps like registration.

Hence we would like to leave room for interfaces for memory allocation to be proposed. For the sake of the proposed interfaces, we assume that communication from virtual memory bears no additional cost. However, we realizes that several networks today do not allow for direct virtual memory communication, they require additional steps such as memory registration.

We would like propose another attribute for discussion in this context

**Communicatable memory attribute** This attribute is used to indicate to the implementation if the memory for either Origin_addr or Target_addr or both of them has been allocated with the above mentioned interfaces for memory allocation. This will allow for additional optimizations at the runtime implementation layer. The characteristics of this attribute are similar to those described in 5.1.1

## 6 Prototype implementation and performance evaluation

We will implement a prototype of these operations within the OpenMPI framework and evaluate performance for both microbenchmarks and application micro-kernels. We will survey the application community on their opinions about usability of these interfaces. Our initial survey included several DOE applications, we received several positive comments about the proposal. We plan to continue these surveys with help from the application community.

## 7 Contributors

Edo Apra (ORNL), Ronald Brightwell (SNL), Richard Graham (ORNL), Robert Harrison (ORNL), Jarek Nieplocha (PNNL), Howard Pritchard (Cray), Galen Shipman (ORNL), Vinod Tipparaju (ORNL), Jeffery Vetter (ORNL)

## References

[1] Ray Barriuso and Allan Knies, "Shmem user's guide for c", Tech. Rep., Cray Research, Inc, 1994.

[2] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda, "High Performance Remote Memory Access Communication: The Armci Approach", *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 233–253, 2006.

[3] Dan Bonachea, "Gasnet specification, v1.1", Tech. Rep., Berkeley, CA, USA, 2002.