

MPI Ruby with Remote Memory Access

Christopher C. Aycock
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, England
christopher.aycock@comlab.ox.ac.uk

Abstract

Advances in communication for parallel programming have yielded one-sided messaging systems. The MPI bindings for Ruby have been augmented to include the remote memory access functions of MPI-2.

1. Introduction

Parallel programming is generally considered to be more difficult than sequential programming. One reason is that in distributed-memory systems the user traditionally schedules the communication himself. MPI-2, the update to the standard Message-Passing Interface, introduced *remote memory access* (RMA) [1] that allows the user to issue *one-sided* communications. With MPI-2, the user is no longer required to pair *sends* and *receives*; he simply directs a process when and where to *put* or *get* data. The MPI bindings for the Ruby scripting language [2] have been updated to support these RMA features.

2. Remote Memory Access

In a system that supports remote memory access, either directly in hardware or through emulation, a process's data may be ubiquitously updated. The result is greater flexibility for the user, particularly in cases where communication is dynamic and may not be known until runtime. Because this one-sided communication in MPI-2 is non-blocking, the communication is not guaranteed to have completed until the processes perform a synchronization, the most common being a collective barrier known as a *fence*.

When the user wants to move data from one process to another, he issues a request to either *put* data into or *get* data from a process's address space. From the user's perspective, the remote process is not aware of

the data transfer. Because these communications are non-blocking, there is no possibility of deadlock.

In order to define exactly what regions of memory are accessible by another process, the user must indicate a "window" of available memory. It is this memory that is written to or read from.

3. Ruby, MPI, and RMA

Ruby is an object-oriented scripting language. Its MPI bindings take advantage of Ruby's dynamic capabilities. There is no need for the user to specify the type or size of the data. In keeping with that tradition, the added RMA features revolve around a Ruby array, rather than a buffer as in C. The user places or extracts data from a specific index of the remote array.

To begin, the user establishes a window by specifying an initial array (which may be empty as represented by []) and an MPI communicator. The user then invokes synchronization to activate an object representing the window. Because Ruby is object-oriented, all RMA functions are members of this window object. Note that `create()` and `fence()` are both collective, although the initial array for `create()` may be different for each process.

```
win = MPI::Win.create(array, comm)
# creates window object

win.fence
# synchronizes the processes

win.object
# returns the array
```

To perform communication, the user specifies the appropriate action with regard to the index of a particular process's array. The semantics for the communication functions mimic those of regular Ruby arrays.

Because Ruby arrays are dynamic and heterogeneous, there is no restriction as to where data may be stored, or what the size or type of the data must be. If the index provided by the user is greater than the index of the final element in the array, then the array will increase in size to accommodate the data. Furthermore, the user may provide data of differing types to different elements of the array.

To remotely place a Ruby object, the user calls the `put()` method of the window object and passes the Ruby object along with the destination and index. Similarly, to obtain remote data, the user calls `get()` with the source and index. However, because RMA procedures are not guaranteed to have completed until after synchronization, the `get()` method returns a “get request” object; the user may query this object after `fence()`.

```
win.put(obj, destination, index)
# destination's array[index] = obj

req = win.get(source, index)
# requests source's array[index]

win.fence
# guarantee completion

req.object
# returns 'get request' object
```

Because `put()` merely places the data, MPI-2 has a function used to perform a remote computation. This function, called `accumulate()`, is similar to `put()` except that an MPI operation is also provided in the argument list. As with the traditional MPI semantics, any of the standard operators may be used; however, only the standard-defined operators are allowed. No user-defined operators may be used with `accumulate()`. However, many user-defined types with overloaded operators may be used.

```
win.accumulate(obj, dest, idx, op)
# dest's array[idx] =
#   obj op array[idx]
```

It is important to note that, as with the traditional MPI semantics, using the same index more than once between synchronizations leads to a race condition. The only exception is `accumulate()`, in which the operations are guaranteed to be atomic. However, in no case is there a guarantee as to the order of completion for communication.

4. Example

To demonstrate the expressiveness of MPI Ruby’s RMA functions, here is an example in which four processes write to an arbitrary root. The data ranges from an integer, to another array, to even a hash table. Ruby takes care of the serialization automatically, so there is no need to bother with marshaling the data for communication.

```
world = MPI::Comm::WORLD
pid = world.rank

win = MPI::Win.create([], world)
win.fence

root = 0
if ( pid == 0 )
  win.put(1, root, 0)
elsif ( pid == 1 )
  win.put([4, 8], root, 2)
elsif ( pid == 2 )
  win.put({"text"=>5}, root, 3)
elsif ( pid == 3 )
  win.put(7, root, 5)
end

win.fence

# win.object on the root is now:
#   [1, nil, [4, 8],
#   {"text"=>5}, nil, 7]
```

5. Conclusion

Parallel programming in distributed-memory systems has traditionally been tedious because of the communication scheduling in two-sided messaging. The remote memory access of MPI-2 combined with the dynamic expressiveness of Ruby create an environment that greatly simplifies parallel programming.

References

- [1] W. Gropp et al. *MPI—The Complete Reference: Volume 2, The MPI Extensions*. MIT Press, 1998.
- [2] E. Ong. MPI Ruby: Scripting in a Parallel Environment. *Computing in Science and Engineering*, 4(4):78–82, 2002.