Leveraging MPI's One-Sided Communication Interface for Shared-Memory Programming

Torsten Hoefler,[1,2] James Dinan,[3] Darius Buntinas,[3]
Pavan Balaji,[3] Brian Barrett,[4] Ron Brightwell,[4]
William Gropp,[1] Vivek Kale,[1] Rajeev Thakur[3]

[1] University of Illinois, Urbana, IL, USA
{htor, wgropp, vivek}@illinois.edu
[2] Department of Computer Science, ETH Zurich, Switzerland
htor@inf.ethz.ch
[3] Argonne National Laboratory, Argonne, IL, USA
{dinan, buntinas, balaji, thakur}@mcs.anl.gov
[4] Sandia National Laboratories, Albuquerque, NM, USA
{bwbarre, rbbrigh}@sandia.gov

Abstract. Hybrid parallel programming with MPI for internode communication in conjunction with a shared-memory programming model to manage intranode parallelism has become a dominant approach to scalable parallel programming. While this model provides a great deal of flexibility and performance potential, it saddles programmers with the complexity of utilizing two parallel programming systems in the same application. We introduce an MPI-integrated shared-memory programming model that is incorporated into MPI through a small extension to the one-sided communication interface. We discuss the integration of this interface with the upcoming MPI 3.0 one-sided semantics and describe solutions for providing portable and efficient data sharing, atomic operations, and memory consistency. We describe an implementation of the new interface in the MPICH2 and Open MPI implementations and demonstrate an average performance improvement of 40% to the communication component of a five-point stencil solver.

1 Introduction

MPI [1] has been the dominant parallel programming model since the mid-1990s. One important reason for this dominance has been its ability to deliver portable performance on large, distributed-memory massively parallel processing (MPP) platforms, large symmetric multiprocessing (SMP) machines with shared memory, and hybrid systems with tightly coupled SMP nodes. For the majority of these systems, applications written with MPI were able to achieve acceptable performance and scalability. However, recent trends in commodity processors, memory, and networks have created the need for alternative approaches. The number of cores per chip in commodity processors is rapidly increasing, and memory capacity and network performance are not able to keep up the same pace. Because memory capacity per core is decreasing, mapping a single operating system process to an MPI rank and assigning a rank per core severely limit

the problem size per rank. In addition, MPI's single-copy model for both message passing and one-sided communication exacerbate the memory bandwidth problem by using intranode memory-to-memory copies to share data between ranks. Moreover, network interfaces are struggling to support the ability for all cores on a node to use the network effectively. As a result, applications are moving toward a hybrid model mixing MPI with shared-memory models that attempt to overcome these limitations [2, 3].

A relatively straightforward and incremental approach to extending MPI to support shared memory has recently been approved by the MPI Forum. Several functions were added, which enable MPI ranks within a shared memory domain to allocate shared memory for direct load/store access. The ability to directly access a region of memory shared between ranks is more efficient than copying and reduces stress on the memory subsystem. Sharing a region of memory between ranks also overcomes the per core memory capacity issue and provides more flexibility in how the problem domain is decomposed. This approach reduces the amount of memory consumed for some data structures such as read-only databases that replicate state across all ranks. From a programming standpoint, providing shared memory supports structured programming, where data is private until it is explicitly shared. The alternative, where data is shared and must be explicitly made private, introduces more complexity into an existing MPI application and the associated MPI implementation. Shared memory is also nearly ubiquitous, given the prevalence of multicore processors.

This paper describes these recent extensions to the MPI Standard to support shared memory, discusses implementation options, and demonstrates the performance advantages of shared memory for a stencil benchmark.

Motivation and Related Work

Support for shared memory in MPI has been considered before, but a number of factors have made such support increasingly compelling. In particular, although POSIX shared memory can be used independently from MPI, the POSIX shared-memory model has several limitations that can be overcome by exposing it through MPI. First, POSIX shared-memory allocation is not a collective operation. One process creates a region of memory and allows other processes to attach to it. Making shared-memory creation collective offers an opportunity to optimize the layout of the memory based on the layout of the ranks. Since the MPI implementation has knowledge of the layout of the shared-memory region, it may be able to make message-passing operations using this region more efficient. For example, MPI may be able to stripe messages over multiple network interfaces, choosing the interface that is closest to the memory being sent. Integration between the MPI runtime system and shared memory simplifies shared-memory allocation and cleanup. Relying on an application using POSIX shared memory directly to clean up after abnormal termination has been problematic. Having the MPI implementation be responsible for allocating and freeing shared memory is a better solution. Knowledge of shared memory inside the MPI implementation also provides better support and integration with MPI

tools, such as correctness and performance debuggers. Furthermore, nearly all MPI implementations already have the infrastructure for allocating and managing shared memory since it is used for intranode data movement, so the burden on existing implementations is light.

Previous work on efficiently supporting MPI on shared-memory systems has concentrated mostly on mapping an MPI rank to a system-level or user-level thread [4–7]. This approach allows MPI ranks to share memory inside an operating system process, but it requires program transformation or knowledge on the part of the programmer to handle global and static variables appropriately. Systems specifically aimed at mixing MPI and shared memory have been developed, effectively augmenting MPI with shared-memory capabilities as the new extensions do. LIBSM [8] and the Unified Parallel System [9] are two such systems developed to support the ability for applications to use both MPI and shared memory efficiently. However, neither of these systems actually made internal changes to the MPI implementation; rather, they provided an application-level interface that abstracted the capabilities of message passing and shared memory.

The need for shared memory in MPI was brought up at the Forum by R. Brightwell, who proposed a malloc/free interface which did not define synchronization semantics. T. Hoefler later proposed to merge this functionality into the newly revamped one-sided communication interface. Hoefler and J. Dinan brought forward a concrete proposal, which the Forum eventually voted for inclusion in MPI-3. The interface described in this paper is what will be included in MPI-3.

2 Extending MPI with Integrated Shared Memory

MPI's remote memory access (RMA) interface defines one-sided communication operations, data consistency, and synchronization models for accessing memory regions that are exposed through MPI windows. The MPI-2 standard defined conservative, but highly portable semantics that would still guarantee correct execution on systems without a coherent memory subsystem. In this model, the programmer reasons about the data consistency and visibility in terms of separate private (load/store access) and public (RMA access) copies of data exposed in the window.

The MPI-3 RMA interface extends MPI-2's *separate* memory model with a new *unified* model, which provides relaxed semantics that can reduce synchronization overheads and allow greater concurrency in interacting with data exposed in the window. The unified model was added in MPI-3 RMA to enable more efficient one-sided data access in systems with coherent memory subsystems. In this model, the public and private copies of the window are logically identical, and updates to either "copy" automatically propagate. Explicit synchronization operations can be used to ensure completion of individual or groups of operations.

The unified memory model defines an efficient and portable mechanism for one-sided data access, including the needed synchronization and consistency op-

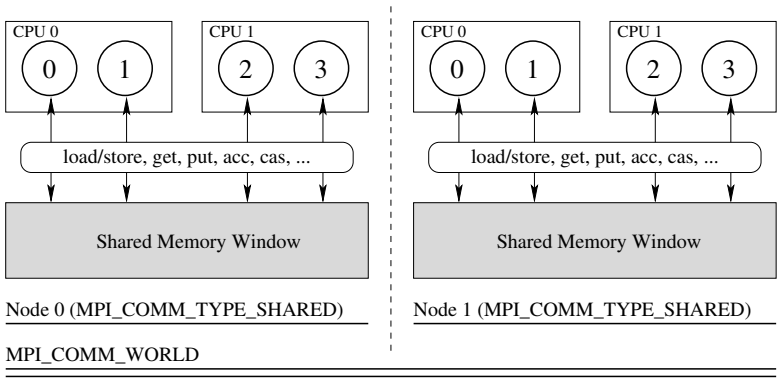


Fig. 1. Interprocess shared-memory extension using MPI RMA; an execution with two nodes is shown, and a shared memory window is allocated within each node.

erations. We observe that this infrastructure already provides several important pieces of functionality needed to define a portable, interprocess shared-memory interface. We now discuss the additional functionality, illustrated in Figure 1, that is needed to extend the RMA model in order to support load/store accesses originating from multiple origin processes to data exposed in a window. In addition, we discuss new functionality that is needed to allow the user to query system topology in order to identify groups of processes that communicate through shared memory.

2.1 Using the RMA Interface for Shared Memory

In the MPI-2 one-sided communication interface, the user first allocates memory and then exposes it in a window. This model of window creation is not compatible with the interprocess shared-memory support provided by most operating systems, which require the use of special routines to allocate and map shared memory into a process's address space. Therefore, we have created a new routine, MPI_Win_allocate_shared, that collectively allocates and maps shared memory across all processes in the given communicator.

CPU load and store instructions are similar to one-sided get and put operations. In contrast with get/put, however, load/store operations do not pass through the MPI library; and, as a result, MPI is unaware of which locations were accessed and whether data was updated. Therefore, the separate memory model conservatively defines store operations as updating to full window in order to prevent data corruption on systems whose memory subsystem is not coherent. However, an overwhelming majority of parallel computing systems do provide coherent memory, and on these systems this semantic is unnecessarily restrictive. Therefore, MPI-3 defines a unified memory model where store operations do not conflict with accesses to other locations in the window. This model closely matches the shared-memory programming model used on most systems, and windows allocated by using MPI_Win_allocate_shared are defined to use the unified memory model.

2.2 Mapping of Inter-Process Shared Memory

Each rank in the shared-memory window provides an allocation size, and a shared memory segment of at least the sum of all sizes is created. Specifying a per rank size rather than a single, global size allows implementations to optimize data locality in nonuniform memory architectures. By default, the allocated shared-memory region is required to be contiguous. That is, the memory region associated with rank N in a given window must be directly before the memory region associated with rank $N + 1$. The info key alloc_shared_noncontig allows the user to relax this allocation constraint. When this key is given, MPI can map the segments belonging to each process into noncontiguous locations. This can enable better performance by allowing MPI to map each segment on a page boundary, potentially eliminating negative cache and NUMA effects.

Many operating systems make it difficult to ensure that shared memory is allocated at the same virtual address across multiple processes. The MPI one-sided interface, which encourages the dynamic creation of shared-memory regions throughout an application's life, exacerbates this problem. MPI_Win_allocate_shared does not guarantee the same virtual address across ranks, and it returns only the address of the shared-memory region for the local rank. MPI_Win_shared_query provides a query mechanism for determining the base address in the current process and size of another process's region in the shared-memory segment. The address of the absolute beginning of the window can be queried by providing MPI_PROC_NULL as the rank argument to this function.

2.3 Querying Machine Topology

The MPI_Win_allocate_shared function expects the user to pass a communicator on which a shared-memory region can be created. Passing a communicator where this is not possible is erroneous. In order to facilitate the creation of such a "shared memory capable" communicator, MPI-3 provides a new routine, MPI_Comm_split_type. This function is an extension of the MPI_Comm_split functionality, with the primary difference being that the user passes a type for splitting the communicator instead of a color. Specifically, the MPI-3 standard defines the type MPI_COMM_TYPE_SHARED, which splits a communicator into subcommunicators on which it is possible to create a shared-memory region.

The MPI_Comm_split_type functionality also provides an info argument that allows the user to request for architecture-specific information that can be used to restrict the communicator to span only a NUMA socket or a shared cache level, for example. While the MPI-3 standard does not define specific info keys, most implementations are expected to provide NUMA and cache management capabilities through these info keys.

3 Implementation of Shared-Memory RMA

The shared-memory RMA interface has been implemented in both MPICH and Open MPI by using similar techniques. In this section we describe the steps

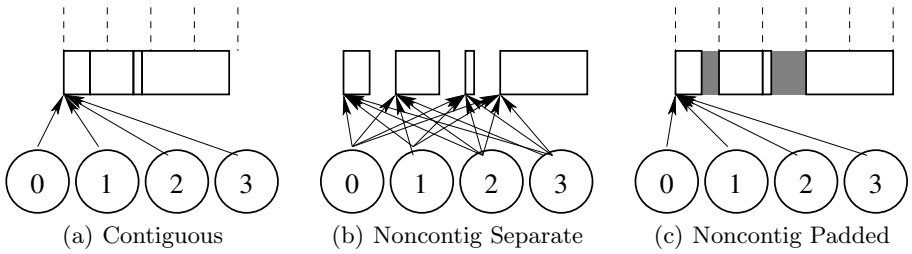
required for the MPI library to allocate a shared window; we also provide implementation details.

The *root* (typically the process with rank 0 in the associated communicator) allocates a shared-memory region that is large enough to contain all of the window segments of all processes sharing the window. Once the shared-memory region has been created, information identifying the shared-memory region is broadcast to the member processes, which then attach to it. At any process, the base pointer of a window segment can be computed by knowing the size and base pointer of the previous window segment: the base pointer of the first window segment, segment 0, is the address of where the shared-memory segment was attached; and the base pointer of segment i is $base\_ptr_i = base\_ptr_{i-1} + seg\_size_{i-1}$.

Scalability needs to be addressed for two implementation issues: (1) computing the sum of the shared window segments in order to determine the size of the shared-memory segment and (2) computing the base pointer of a window segment. For windows with a relatively small number of processes, an array of the segment size of each process can be stored locally at each process by using an all-gather operation. From this array, the root process can compute the size of the shared-memory segment, and each process can compute the base pointer of any other segment. For windows with a large number of processes, however, the offsets may be stored in a shared-memory segment, with scalable collectives (reduce, broadcast, exscan) used to compute sizes and offsets.

When the alloc_shared_noncontig info key is set to "true," the implementation is not constrained to allocate the window segments contiguously; instead, it can allocate each window segment so that its base pointer is aligned to optimize memory access. Individual shared-memory regions may be exposed by each rank, an approach that can be used to provide optimal alignment and addressing but requires more state. An alternative implementation would be to allocate the window as though it was allocated contiguously, except that the size of each window segment is rounded up to a page boundary. In this way each window segment is aligned on a page boundary, and shared state can be used to minimize resource utilization. Both MPICH and Open MPI use the latter approach.

Figure 2 shows the three shared-memory allocation strategies discussed above. In Figure 2(a) we see the contiguous memory allocation method. The figure



(a) Contiguous (b) Noncontig Separate (c) Noncontig Padded

Fig. 2. Shared-memory window allocation strategies. Dotted lines in (a) and (c) represent page boundaries. In (b) each window segment is allocated in a separate shared-memory region and is page aligned.

shows four processes each of which has the entire memory region attached. The shared-memory region contains four window segments of different sizes. Figures 2(b) and 2(c) show noncontiguous allocations. In Figure 2(b) each window segment is allocated in a separate shared-memory region. Each process attaches all the memory regions. In Figure 2(c) a single shared-memory region is attached by each process. Each window segment is padded out to a window boundary. The first and third segments do not end on a page boundary; thus, we see that those segments are padded so that the next window segment starts on a page boundary.

Synchronization operations must provide processor memory barriers to ensure consistency semantics but otherwise are straightforward to implement. Because of the direct memory access available for all target operations, communication calls may be implemented as memory copies performed during the communication call itself. While an implementation could choose to implement the accumulate operations by using processor atomics, locks and memory copies can also provide the required semantics. Both MPICH and Open MPI use a spinlock per target memory region to implement accumulate operations, because of the simplicity of implementation and greater portability.

4 Use Cases and Evaluation

Shared-memory windows in MPI programs have multiple effects on future parallel programming techniques. Current scientific applications often use OpenMP to enable sharing of large data structures (e.g., hash tables or lookup tables/-databases) among cores inside a compute node. This approach requires using two different models of parallelization: MPI and a carefully crafted OpenMP layer that enables scalability to the large core counts (32–64) in today's architectures. This often requires an "MPI-style" domain decomposition of the OpenMP parts, effectively leading to a complex two-stage parallelization of the program. Shared-memory windows allow a structured approach to this issue in that OpenMP can be used where it is most efficient (e.g., at the loop level) and shared memory can be shared across different MPI processes with a single level of domain decomposition.

A second use-case is to use shared-memory windows for fast intranode communications. Here, the user employs a two-level parallelization in order to achieve the highest possible performance using true zero-copy mechanisms (as opposed to MPI's mandated single-copy from send buffer to receive buffer). This has the advantage over a purely threaded approach that memory is explicitly shared and heap corruptions due to program bugs are less likely (cf. [10]). An example of this benefit explored with an early prototype of the shared-memory extensions can be found in [11]. This work demonstrates the incremental approach of incorporating shared memory into an MPI application in order to reduce the iteration count of the linear solver portion of an application. The rest of the application, which performs and scales well, can remain unchanged and largely unaware of the use of shared memory.

Shared-memory regions can also help better support the use of accelerators within an MPI application. For example, if an application is running with one MPI rank per core and all ranks wish to transfer data to a GPU, it can be challenging to coordinate the transfer of data between the host memory of each rank and GPU memory. Using shared memory, one rank can be responsible for transferring data between the host and the device, reducing the amount of coordination among ranks.

Five-Point Stencil Kernel Evaluation

We will now evaluate the performance improvements that can be achieved with shared-memory windows using an application kernel benchmark. We prefer not to show the usual ping-pong benchmarks because they would simply show the MPI overhead versus the performance of the memory subsystem while hiding important effects caused by the memory allocation strategy. Instead, we use a simple, two-dimensional Poisson solver, which computes a heat propagation problem using a five-point stencil. The $N \times N$ input grid is decomposed in both dimensions by using `MPI_Dims_create` and `MPI_Cart_create`. The code adds one-element-deep halo zones for the communication. The benchmark utilizes nonblocking communication of $8 \cdot N$ Bytes in each direction to update the halo zones and `MPI_Waitall` to complete the communication. It then updates all local grid points before it proceeds to the next iteration.

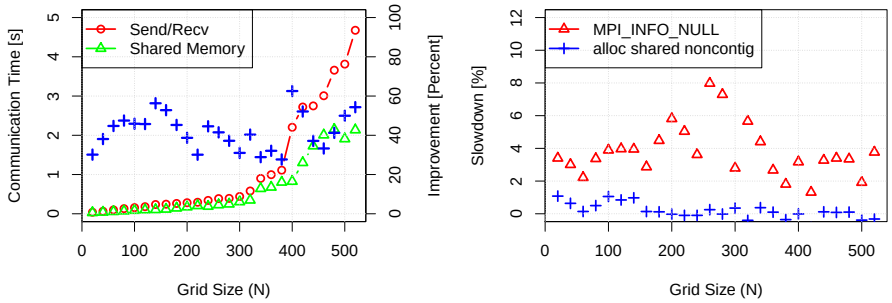
The shared-memory implementation utilizes `MPI_Comm_split_type` to create a shared-memory communicator and allocates the entire work array in shared memory. Optionally, it provides the `alloc_shared_noncontig` info argument to allow the allocation of localized memory. The communication part of the original code is simply changed to `MPI_Win_fence` in order to ensure memory consistency and direct memory copies from remote to local halo zones. To simplify the example code, we assume that all communications are in shared memory only. The following listing shows the relevant parts of the code (variable declarations and array swapping are omitted for brevity).

```
MPI_Comm_split_type(comm, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &shmcomm);

MPI_Win_allocate_shared(size*sizeof(double), info, shmcomm, &mem, &win);
MPI_Win_shared_query(win, north, &sz, &northptr);
// ... south, east, west directions

for(iter=0; iter<niters; ++iter) {
  MPI_Win_fence(0, win); // start new access and exposure epoch
  if(north != MPI_PROC_NULL) // the "communication"
    for(int i=0; i<bx; ++i) a2[ind(i+1,0)] = northptr[ind(i+1,by)];
  // ... south, east, west directions
  update_grid(&a1, &a2); // apply operator and swap arrays
}
```

We ran the benchmark on a six-core 2.2 GHz AMD Opteron CPU with two MPI processes and recorded communication and computation times separately.

Open MPI and MPICH perform similarly because of the similar implementations; we focus on experimentation with the MPICH implementation.

Figure 3(a) shows the communication times of the send/recv version (red line with dots) and the shared-memory window versions (green line with triangles), as well as the communication time improvement of the shared-memory window version (blue crosses). In general, we show that the communication overhead for shared-memory window version is 30-60% lower than for the traditional message-passing approach. This is due to the direct memory access and avoided matching queue and function call costs.



(a) Communication Overhead (b) Computation Slowdown

Fig. 3. Communication and computation performance for the five-point stencil kernel.

Figure 3(b) shows the computation time of the shared-memory window version, that is, the time to update the inner grid cells relative to the computation time of the send/recv version. We observe a significant slowdown (up to 8%) of the computation without the alloc_shared_noncontig argument. This is partially due to false sharing and the fact that the memory is local to rank 0. Indeed, the slowdown of the computation eliminated any benefit of the faster communication and made the parallel code slower. Specifying alloc_shared_noncontig eliminates the overhead down to the noise ($< 1.7\%$) and leads to an improvement of the overall runtime.

5 Conclusions and Outlook

In this work, we described an MPI standard extension to integrate shared memory functionality into MPI-3.0 through the remote memory access interface. We motivated this new interface through several use-cases where shared memory windows can result in improved performance, scaling, and capabilities. We discussed the design space for this new functionality and provided implementations in two major MPI implementations which will both be available shortly in the official releases.

To evaluate the application-level impact of shared memory windows, we performed a performance study using a heat-propagation 5-point stencil benchmark.

The benchmark illustrated two important aspects: (1) an average 40% reduction in data movement time compared with a traditional send/recv formulation and (2) the potentially detrimental slowdown of computation if false sharing and NUMA effects are ignored. By allowing the MPI implementation to automatically adjust the shared memory mapping, we showed that these negative performance effects can be eliminated.

For future work, we plan to further investigate NUMA-aware allocation strategies, direct mapping of shared memory (e.g., XPMEM), and the effective use of the info argument to MPI_Comm_split_type to expand this routines topology querying capabilities. We also plan to apply the shared memory extensions to incomplete factorization codes, as well as to a human heartbeat simulation code.

Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357, under award number DE-FC02-10ER26011 with program manager Sonia Sachs, under award number DE-FG02-08ER25835, and as part of the Extreme-scale Algorithms and Software Institute (EASI) by the Department of Energy, Office of Science, U.S. DOE award DE-SC0004131. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energys National Nuclear Security Administration, under contract DE-AC-94AL85000.

References

1. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4 2009) `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.
2. Smith, L., Bull, M.: Development of mixed mode MPI / OpenMP applications. Scientific Programming **9**(2,3) (2001) 83–98
3. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proc. of the 17th Euromicro Intl. Conf. on Parallel, Distributed and Network-based Processing. (Feb. 2009)
4. Demaine, E.: A threads-only MPI implementation for the development of parallel programs. In: Proceedings of the 11th Intl. Symp. on HPC Systems. (1997) 153–163
5. Bhargava, P.: MPI-LITE: Multithreading Support for MPI. (1997) `http://pcl.cs.ucla.edu/projects/sesame/mpi_lite/mpi_lite.html`.
6. Shen, K., Tang, H., Yang, T.: Adaptive two-level thread management for fast MPI execution on shared memory machines. In: Proceedings of the ACM/IEEE Conference on Supercomputing. (1999)
7. Tang, H., Shen, K., Yang, T.: Program transformation and runtime support for threaded MPI execution on shared memory machines. ACM Transactions on Programming Languages and Systems **22** (2000) 673–700
8. Shirley, D.: Enhancing MPI applications through selective use of shared memory on SMPs. In: Proc. of the 1st SIAM Conference on CSE. (September 2000)
9. Los Alamos National Laboratory: Unified Parallel Software Users' Guide and Reference Manual. (2001)
10. Lee, E.A.: The problem with threads. Computer **39**(5) (May 2006) 33–42
11. Heroux, M.A., Brightwell, R., Wolf, M.M.: Bi-modal MPI and MPI+threads computing on scalable multicore systems. IJHPCA (Submitted) (2011) .