Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters*

Weihang Jiang[1], Jiuxing Liu[1], Hyun-Wook Jin[1], Dhabaleswar K. Panda[1], Darius Buntinas[2], Rajeev Thakur[2], and William Gropp[2]

[1] Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210
{jiangw, liuj, jinhy, panda}@cis.ohio-state.edu
[2] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
{buntinas,thakur,gropp}@mcs.anl.gov

Abstract. In this paper we compare various design alternatives for synchronization in MPI-2 passive one-sided communication on InfiniBand clusters. We discuss several requirements for synchronization in passive one-sided communication. Based on these requirements, we present four design alternatives, which can be classified into two categories: thread-based and atomic operation-based. In thread-based designs, synchronization is achieved with the help of extra threads. In atomic operation-based designs, we exploit InfiniBand atomic operations such as Compare-and-Swap and Fetch-and-Add. Our performance evaluation results show that the atomic operation-based design can require less synchronization overhead, achieve better concurrency, and consume fewer computing resources compared with the thread based design.
Keywords: MPI-2, Passive target communication, Synchronization, InfiniBand, Cluster

1 Introduction

MPI has been the de facto standard in high-performance computing for writing parallel applications. As an extension to MPI, MPI-2 [11] introduces several new features. One important new feature is one-sided communication. In the traditional two-sided communication, both parties must perform matching communication operations (e.g., a *send* and a *receive*). In one-sided operation, a matching operation is not required from the remote party. All parameters for the operation, such as source and destination buffers, are provided by the initiator of the operation. One-sided operations can support more flexible communication patterns and improve performance in certain applications.

* This research is supported by Department of Energy's grant #DE-FC02-01ER25506, National Science Foundation's grants #CCR-0204429 and #CCR-0311542, and Post-doctoral Fellowship Program of Korea Science & Engineering Foundation(KOSEF). This work is also supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

In MPI-2 one-sided communication, the process that initiates the operation is called the *origin* process, and the process being accessed is called the *target* process. The memory area in the target process that can be accessed is called the *window*. MPI-2 requires explicit synchronization to guarantee the completion of data communication operations on windows. MPI-2 supports two synchronization modes: active and passive. In active mode, the target is actively involved in synchronization, whereas in passive mode, the target is not explicitly involved in synchronization.

We have previously implemented MPI-2 active mode one-sided communication in [7] using RDMA-based communication over InfiniBand. In this paper we extend this implementation to allow passive one-sided communication. The main challenge in extending active mode to passive mode is designing an efficient synchronization mechanism that allows one-sided operations to be performed at the target node independently of what the target process is doing.

Several design challenges are involved in implementing efficient MPI-2 passive one-sided communications. First, the implementation must be able to make independent progress in passive one-sided communications. Next, concurrent communications must be handled efficiently. Another issue is efficient implementation of shared and exclusive locks. A related issue is how to implement MPI_Win_lock() in a nonblocking fashion. Further, since modern network interfaces, such as InfiniBand, offer RDMA-based operations, these operations should be used in the most efficient manner.

In this paper we take on these challenges. We implement and evaluate four design alternatives for passive-mode synchronization: dedicated thread, event-driven thread, Test-and-Set, and MCS based designs.

All these designs have been incoperated in our MPI-2 implementation over InfiniBand, MVAPICH2 [14] [9], which is based on MPICH2 developed by Argonne National Laboratory [1].

The paper is organized as follows. In Section 2, we briefly describe InfiniBand. In Sections 3–5, we present design issues. In Section 6, we evaluate the performance of our various designs. In Section 7, we discuss related work. In Section 8, we draw conclusions and discuss future work.

2 InfiniBand and Atomic Operations

The InfiniBand Architecture is an industry standard for high-performance interconnects between processing nodes and I/O nodes [5]. Host channel adapters (HCAs) connect nodes to the InfiniBand fabric. InfiniBand provides both channel and memory semantics. In channel semantics, send and receive are used for communication. In memory semantics, InfiniBand supports remote direct memory access (RDMA) operations.

Another emerging feature of InfiniBand is remote atomic operation. This allows us to efficiently implement the synchronization algorithms designed for a shared-memory environment to a distributed private-memory environment. InfiniBand supports two 64-bit atomic operations: Compare-and-Swap and Fetch-and-Add. The Compare-and-Swap operation reads a 64-bit content from the

memory of a remote process, compares the content with the *compare_value* parameter of this atomic operation, and puts the value of the *swap_value* parameter into the remote memory if the two compared values are the same. The Fetch-and-Add operation reads data from the remote memory, performs an addition operation between the data and the *add_value* parameter of this atomic operation, and updates the result to the remote memory. Both the Compare-and-Swap and Fetch-and-Add operations bring back the old value of the variable in the remote memory. As the name "atomic operation" suggests, Compare-and-Swap and Fetch-and-Add operations are handled atomically, and more important, they are processed by the processor on the HCAs, without CPU involvement at the remote side.

3 Design Issues in Passive One-Sided Communication

In this section we discuss challenges in implementing efficient MPI-2 passive one-sided communication.

Synchronization Performance: When the contention for synchronization functions is low, low synchronization overhead is important. When the contention is high, low synchronization delay is desirable. In both cases, the number of messages exchanged for synchronization functions should be small. **Independent Progress:** In MPI-2 passive one-sided communication, the target process does not make any MPI calls to cooperate with the origin process for communication or synchronization. Therefore, the implementation cannot rely on the progress engine being called by MPI functions, the strategy commonly used for two-sided communication. **Concurrent Communication:** In MPI-2, one-sided communication and two-sided communication may happen concurrently. One-sided communication from different origin processes to disjoint windows at a target process may also happen concurrently. Hence we need to handle this situation in our design. **Nonblocking MPI_Win_lock():** When MPI_Win_lock() is called, if the lock at the target process is held by other processes, the lock cannot be acquired until the lock is released. The nonblocking MPI_Win_lock() allows the current process to continue without waiting for the lock. However, the lock must be acquired before the first communication operation takes effect. **Shared Locks and Exclusive Locks:** In MPI-2, both shared locks and exclusive locks are supported. **RDMA for Data Transfer:** Modern network interfaces, such as InfiniBand, offer RDMA-based operations, which can be used for high-performance data transfer in one-sided communication [7]. Therefore, synchronization mechanisms must work correctly with RDMA-based communication.

In following sections, we address these challenges and describe thread-based and atomic operation-based designs for MPI-2 passive mode synchronization.

4 Thread-Based Designs

Thread-based design is widely used in MPI implementations to support one-sided communication. In our MPI-2 implementation over InfiniBand, we use RDMA operations to transfer data, while we still use a thread running at the target process to handle special cases such as noncontiguous data transfer and

the accumulate function. Therefore, we are interested in whether we can use this thread to handle passive synchronization efficiently.

4.1 Dedicated Thread-Based Design

In a dedicated thread-based design, an assisting thread runs at the target side in a dedicated manner and handles all passive synchronization requests from the origin processes. In order to achieve low latency, the thread is always active and uses polling to process communication. The characteristics of the thread guarantee the independent progress of the synchronization process. Before using RDMA operations to transfer data, we need to acquire the lock. This is done by sending a control message and getting an acknowlegement back. In order to implement MPI_Win_lock() in a nonblocking manner, after sending the lock request, the origin process continues its work, buffering any one sided operations until the lock is acquired. Since in a thread-based design, the target process knows whether the request message is for a shared lock or an exclusive lock, the target process can coordinate between shared-lock requests and exclusive-lock requests to support both of them. In this implementation, passive synchronization messages and two-sided messages are handled by different threads. Eventually the MPICH2 internal progress engine will be thread safe, until then, we need to provide our own mutual exclusion mechanisms to serialize access by the two threads.

The dedicated thread-based design can achieve good portability because it is based on two-sided message passing, which is supported by all platforms. However, there are some shortcomings in its performance. One problem is related to concurrent communication. In addition to concurrent two-sided communication, one-sided communication can also happen concurrently with other one-sided communication. When there are multiple windows at a target process, different origin processes may access them; hence, using a single thread does not allow concurrent accesses. A simple solution is to use multiple threads, one thread per window. However, in such thread-based designs, as long as all the threads are sharing the same progress engine, the time spent on the progress engine cannot be overlapped, because of the mutexes controlling access to the progress engine. Therefore, compared with the dedicated thread-based design, using multiple threads can not improve performance. Performance with multiple threads is discussed in Section 6. Another problem is that since the dedicated thread keeps running at a target process, CPU cycles are consumed by the thread even if there is no passive one-sided communication. In the next section, we introduce an event-driven based design to solve this problem.

4.2 Event Driven Based Design

InfiniBand provides both channel and memory semantics. By using channel semantics, one process can generate a signal at a remote process. Also, InfiniBand HCA supports event handlers, which can be driven by such a signal.

Combining these features, we propose an event driven-based design to reduce the CPU utilization. Initially, a predefined event handler that can resume a blocked thread is registered to an HCA. When the assisting thread is created,

the process posts a receive operation and blocks the thread. If a process wants to communicate with a remote process, it first posts a send operation matching with the prepost receive operation at the remote party, to generate a signal. Then the event wakes the thread up. The remaining steps are similar to the dedicated thread-based design. Finally, before the thread is blocked again, another receive operation is posted.

In the event driven-based design, a thread can be blocked while idle to reduce the CPU utilization. However, compared with the dedicated thread-based design, extra time is spent on generating the signal and waking up the thread. Performance numbers of the event driven-based design are given in Section 6.

5 Atomic Operation-Based Design

In Section 2, we described hardware-level remote atomic operations in InfiniBand. These give us the opportunity to exploit some well-known algorithms proposed for shared-memory synchronization. In this section, we present two designs based on the Test-and-Set and MCS lock algorithms.

5.1 Test-and-Set-Based Design

In the Test-and-Set algorithm, a flag is used to indicate whether the lock is held. To acquire a lock, a processor tries to change the flag from false to true by executing a Test-and-Set instruction. The processor releases the lock by changing the flag back to true. To implement MPI-2 synchronization functions using the Test-and-Set lock algorithm, we can use the atomic operation Compare-and-Swap.

Since the atomic operations are handled by HCA and the MPI library at the target process is not involved, the progress of passive synchronization is independent of the progress of the target process. Once the Compare-and-Swap for acquiring a lock succeeds, the process can start using RDMA to transfer data. To implement nonblocking MPI_Win_lock, a process issues the first Compare-and-Swap operation in MPI_Win_lock(), without waiting for it to complete. The waiting is delayed until the first communication operation.

We can easily extend the Test-and-Set algorithm to support both shared lock and exclusive lock, by checking the value returned by Compare-and-Swap operation. Details can be found in [6].

One drawback of the Test-and-Set-based design is the high network traffic caused by repeated issue of Compare-and-Swap operations. Using an exponential back-off mechanism can alleviate this problem.

5.2 MCS-Based Design

The MCS algorithm is proposed as a scalable synchronization algorithm for shared-memory multiprocessors [10]. The main idea of MCS is to maintain a distributed queue for processes competing for the lock. Scalability is achieved by avoiding spinning on remote memory, and by decreasing the lock synchronization delay.

For each window, each origin process maintains three data structures — *flag*, *previous*, and *next* — and each target process maintains one data structure, called *lock*. When origin process A requests a lock on the target process, it swaps

its process id with the value of *lock*. Then origin process B requests the same lock by swapping. Based on the value swapped back, origin process B knows that origin process A is queued before it. Thus, it updates the value of *next* in origin process A. When origin process A releases the lock, based on the value of *next*, origin process A updates the value of *flag* in the origin process B. Finally, when the origin process B releases the lock, it resets *lock* to NULL at the target process. The atomic Swap operation is used to update the *lock* value atomically, and RDMA Write is used to update the *next*, and *flag* values.

Compared with other atomic operation-based synchronization algorithms, MCS has an advantage in terms of scalability. However, since InfiniBand does not support an atomic Swap operation, emulating it with repeated Compare-and-Swap operations can increase the number of messages when the number of processes competing for the lock increases. In Section 6, we show that when multiple processes compete for the same lock, the number of messages exchanged in the MCS-based design is no longer constant.

6 Performance Evaluation

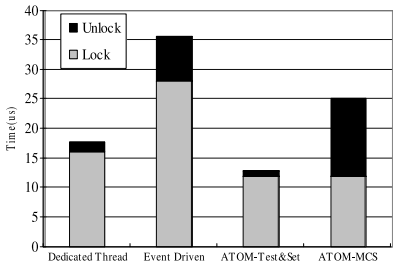

Fig. 1. Synchronization Overhead

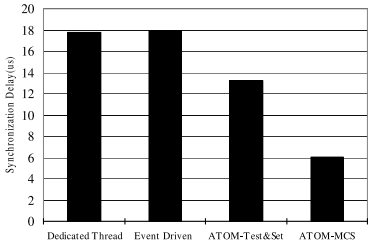

Fig. 2. Synchronization Delay

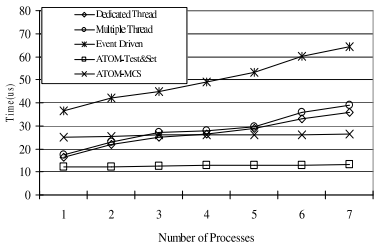

Fig. 3. Concurrency

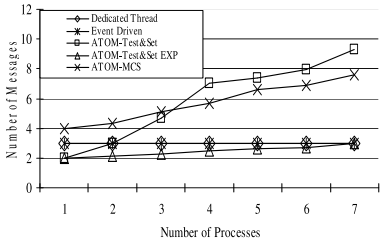

Fig. 4. Message Complexity

The performance of MPI-2 passive synchronization functions can be evaluated with respect to the following metrics: (1) synchronization overhead: time spent on synchronization functions, (2) synchronization delay: time required after one origin process releases a lock on a remote window and another origin process acquires the same lock, (3) concurrency: the capability to handle multiple concurrent passive synchronization functions, (4) message complexity: the number of messages exchanged for synchronization functions, and (5) CPU utilization: the CPU cycles involved in the synchronization process.

Our experimental testbed consists of a cluster of eight SuperMicro SUPER X5DL8-GG nodes, each with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, PCI-X 64-bit 133 MHz bus, and connected to Mellanox InfiniHost MT23108 DualPort 4x HCAs. The nodes are connected by using a Mellanox InfiniScale 24 port switch MTS 2400. The kernel version used is Linux 2.4.22smp. The InfiniHost SDK version is 3.0.1 and the HCA firmware version is 3.0.1. The Front Side Bus (FSB) runs at 533 MHz. The physical memory is 1 GB of PC2100 DDR-SDRAM memory.

6.1 Synchronization Overhead

We begin with a simple approach to measure synchronization overhead. In this test, one process calls only MPI-2 passive synchronization functions (MPI_Win_lock and MPI_Win_unlock) on a window at the other process for multiple iterations. We then report the time taken for each iteration. Figure 1 shows the synchronization overhead for all four designs. We also report separately the time spent acquiring the lock and the time spent releasing the lock. We can see that the Test-and-Set-based design shows the best performance, approximately 12.83 μs. Releasing a lock is faster for this design because processes do not need to wait for the completion of unlock. We also see that using a dedicated thread can achieve better performance than using atomic operations with MCS. The event-driven approach shows the worst performance.

6.2 Synchronization Delay

Synchronization delay is the delay between one origin process releasing a lock on a remote window and another origin process acquiring the same lock. It is an important performance metric for a lock algorithm, especially when the competition between different origin processes for a given lock is heavy. The test for measuring synchronization delay consists of multiple iterations, using two origin processes and one target process. In the even-numbered iterations, origin process 1 requests a lock earlier than does origin process 2, and in the odd-numbered iterations, origin process 2 requests a lock earlier than does origin process 1. After acquiring the lock, each process holds the lock for time E and then releases the lock. The value of E we used is always longer than the synchronization overhead of all designs. As we can see in Figure 2, the designs based on atomic operations outperforms the thread-based designs. The MCS-based design shows the best synchronization delay because locks can be transferred to the next process by using a single message. In all other designs, at least a roundtrip time is required.

6.3 Concurrency

For some MPI-2 applications, the target process may have a large volume of data to be accessed by multiple origin processes. One way to improve the application performance is to use multiple windows and let different origin processes concurrently access the data in different windows. To evaluate how different designs handle concurrent accesses, we used a test with multiple origin processes and one target process. In the target process, multiple windows are created, and the number of windows is equal to the number of origin processes. In each iteration, each origin process calls only MPI_Win_lock and MPI_Win_unlock on the

corresponding target window. We then report the average time spent on each iteration. Figure 3 shows that for Test-and-Set-based design and MCS-based design, the time spent on synchronization functions does not change. This result indicates that they can handle concurrent accesses efficiently. However, for thread-based designs, the time increases when the number of origin processes increases. Further, we can see that even using multiple threads, we cannot achieve better concurrency.

6.4 Message Complexity

Message complexity shows the number of messages exchanged for synchronization functions. A good design should have low message complexity. Our test uses multiple origin processes and one target process. All the origin processes compete for the same lock on a target window. Then we calculate the average number of messages exchanged between one origin process and one target process.

As we can see in Figure 4, with an increase in the number of origin processes, the number of messages of both thread-based designs does not change. For the Test-and-Set-based design, the exponential back-off mechanism reduces the number of messages from 9 to 3 when the number of origin processes is 7. Although the MCS algorithm is proposed as a scalable algorithm, without the atomic Swap operation the number of messages increases to around 7, when the number of origin processes is 7.

6.5 CPU Utilization

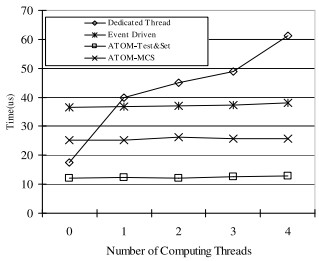

Fig. 5. Computing Thread

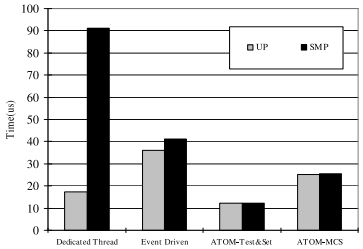

Fig. 6. Assignment Mode

One shortcoming of the thread-based designs is that the assisting thread consumes a significant number of CPU cycles. We evaluate this problem in two scenarios.

Computing Thread: In an MPI-2 application, each process may have multiple computing threads running. We evaluated the performance of synchronization functions under this scenario. Our test uses two processes: one target process and one origin process. The target process spawns several computing threads, and the origin process calls synchronization functions MPI_Win_lock and MPI_Win_lock on a window at the target process for multiple iterations. We then measured the time for each iteration. Figure 5 shows that for both atomic-based designs, the time remains almost unchanged, while for the dedicated thread-based design, the time increases with an increase in the number of computing threads. For the

event-driven based design, since the assisting thread is awakened by a signal, the time almost remains constant, too.

SMP Mode: Until now, all our evaluations have been done in uniprocessor (UP) mode, which means we assign one MPI application process on each node. But many scientific applications are computationally intensive, and running them in *SMP* mode is more suitable. To evaluate this aspect, we proposed a test based on the synchronization overhead test we used before. This SMP test runs two origin processes, A and B, on one dual-CPU workstation and two target processes, C and D, on the other dual-CPU workstation. Origin process A calls synchronization functions to target process C, while origin process B does the same thing to target process D. Figure 6 shows that both atomic operation-based designs can deliver similar performance in both modes. However, the dedicated thread-based design does not work well in one process per processor mode because of the polling of communication threads.

6.6 Discussion

From the performance results we can see that, in general, atomic operation-based designs outperform thread-based designs. By taking advantage of atomic operations in InfiniBand, we can achieve better synchronization overhead and synchronization delay. Atomic operation-based designs can also achieve better concurrency and independent communication progress. One possible drawback of the atomic operation-based design is that the number of messages to acquire a lock increases when there is high contention for the lock. For the Test-and-Set-based design, this problem can be solved by using exponential backoff. For the MCS-based design, the problem stems from the lack of an atomic Swap operation in the current InfiniBand implementation.

7 Related Work

Most implementations of MPI-2 passive one-sided communication are implemented based on the thread-based design [3], [12], [13], [18], [16], [2]. Work in [17], [4] describe a nonthreaded NEC MPI/SX implementation of MPI-2 passive one-sided communication, where the progress engine takes charge of performing synchronization. This design does not satisfy the requirements we have defined in Section 3. The latest version of LAM-MPI [8] supports a part of MPI-2 functions that do not include the passive one-sided communication. Another programming model that allows one to access the memory area of other processes such as MPI-2 one-sided communication is Aggregate Remote Memory Copy Interface (ARMCI) [15], where all synchronization calls involve both communication sides.

8 Conclusions and Future Work

In this paper, we analyzed issues and concerns related to designing a high-performance MPI-2 passive synchronization mechanisms on InfiniBand clusters. We proposed, implemented, and evaluated two thread-based designs (i.e., dedicated thread and event-driven blocking thread-based designs) and two atomic operation-based designs (i.e., Test-and-Set and MCS-based design). We demonstrated that by taking advantage of InfiniBand atomic operations, we can achieve efficient synchronization and deliver good performance.

In the future, we plan to use real applications to study the impact of synchronization in MPI-2 passive communication. We also plan to investigate how to handle datatypes efficiently in MPI-2 one-sided communication.

References

1. Argonne National Laboratory. MPICH2. http://www.mcs.anl.gov/mpi/mpich2/.
2. N. Asai, T. Kentemich, and P. Lagier. MPI-2 Implementation on Fujitsu Generic Message Passing Kernel. In *SC*, 1999.
3. S. Booth and F. E. Mourao. Single Sided MPI Implementations for SUN MPI. In *SC*, 2000.
4. M. Golebiewski and J. L. Traff. MPI-2 One-Sided Communications on a Giganet SMP Cluster. In *EuroPVM/MPI*, 2001.
5. InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.
6. W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, D. Buntinas, R. Thakur, and W. Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication over InfiniBand Clusters. Technical Report OSU-CISRC-5/04-TR34, May 2004.
7. W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, W. Gropp, and R. Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand. In *IEEE/ACM CCGrid*, 2004.
8. LAM Team, Indiana University. LAM 7.0.4.
9. J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *IPDPS*, April 2004.
10. J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. In *ACM Transactions on Computer System*, 1991.
11. Message Passing Interface Forum. MPI-2: A Message Passing Interface Standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.
12. E. Mourao and S. Booth. Single Sided Communications in Multi-Protocol MPI. In *EuroPVM/MPI*, 2000.
13. F. E. Mourao and J. G. Silva. Implementing MPI's One-Sided Communications for WMPI. In *EuroPVM/MPI*, September 1999.
14. Network-Based Computing Laboratory. MVAPICH2: MPI-2 for InfiniBand on VAPI Layer. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html, January 2003.
15. J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. *Lecture Notes in Computer Science*, 1586, 1999.
16. M. Schulz. Efficient Coherency and Synchronization Management in SCI based DSM systems. In *SCI-Europe, Conference Stream of Euro-Par*, 2000.
17. J. Traff, H. Ritzdorf, and R. Hempel. The Implementation of MPI-2 One-Sided Communication for the NEC SX. In *SC*, 2000.
18. J. Worringen, A. Gaer, and F. Reker. Exploiting Transparent Remote Memory Access for Non-Contiguous and One-Sided-Communication. In *CAC*, April 2002.