Natively Supporting True One-sided Communication in MPI on Multi-core Systems with InfiniBand*

G. Santhanaraman[1], P. Balaji[2], K. Gopalakrishnan[1], R. Thakur[2], W. Gropp[3] and D. K. Panda[1]

[1]Computer Science and Engg.,
Ohio State University,
{santhana, gopalakk, panda}@cse.ohio-state.edu

[2]Mathematics and Computer Science,
Argonne National Laboratory,
{balaji, thakur}@mcs.anl.gov

[3]Dept. of Computer Science,
University of Illinois, Urbana Champaign,
wgropp@illinois.edu

Abstract

As high-end computing systems continue to grow in scale, the performance that applications can achieve on such large scale systems depends heavily on their ability to avoid explicitly synchronized communication with other processes in the system. Accordingly, several modern and legacy parallel programming models (such as MPI, UPC, Global Arrays) have provided many programming constructs that enable implicit communication using one-sided communication operations. While MPI is the most widely used communication model for scientific computing, the usage of one-sided communication is restricted; this is mainly owing to the inefficiencies in current MPI implementations that internally rely on synchronization between processes even during one-sided communication, thus losing the potential of such constructs.

In our previous work, we had utilized native one-sided communication primitives offered by high-speed networks such as InfiniBand (IB) to allow for true one-sided communication in MPI. In this paper, we extend this work to natively take advantage of one-sided atomic operations on cache-coherent multi-core/multi-processor architectures while still utilizing the benefits of networks such as IB. Specifically, we present a sophisticated hybrid design that uses locks that migrate between IB hardware atomics and multi-core CPU atomics to take advantage of both. We demonstrate the capability of our proposed design with a wide range of experiments illustrating its benefits in performance as well as its potential to avoid explicit synchronization.

1 *Introduction*

As we advance into an era of petascale sciences, High End Computing (HEC) systems are continuing to meet the requirements of several grand challenge applications. Systems with hundreds of thousands of cores are already available and researchers are looking forward to systems with millions of processors in the next decade. However, the performance that applications can achieve on such large-scale systems depends heavily on their ability to avoid synchronization with other processes, thus minimizing idleness caused by process skew. Towards this goal, scientific applications have traditionally relied on two models for minimizing such synchronization requirements—clique-based communication and implicit data movement using one-sided operations.

Clique-based communication refers to the ability of applications to form small sub-groups of processes with a majority of the communication happening within the groups. Nearest neighbor (e.g., PDE solvers, molecular dynamics simulations) and cartesian grids (e.g., FFT solvers) are popular examples of such communication [4, 12, 5]. While clique-based communication reduces the number of processes each process needs to synchronize with, it does not completely avoid synchronization. Similarly, while the size of the clique grows slowly as compared to the overall system size, on ultra-scale systems, this can still be a concern. For example, in a 2-D cartesian grid communication along a row of processes, on a million process system, each clique can contain as many as a thousand processes.

Implicit data movement using one-sided operations supplements the benefits of clique-based communication by allowing data to be moved from one process' memory to another without requiring any synchronization. Many modern and legacy parallel programming models (e.g., MPI [14], UPC [1], Global Arrays [3]) are increasingly providing constructs for such one-sided communication, where a process can read/write data from another process without necessarily requiring participation from the remote process. While MPI has been the de facto standard for communication on HEC systems, its capability for such implicit one-sided communication is limited, as compared to the UPC and Global Arrays models. This limitation is primarily due to the inefficiencies of current MPI implementations. Specifically, current MPI implementations internally rely on synchronization between processes even during one-sided communication, thus limit-

*This work was supported in part 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 DE-AC02-06CH11357, in part by #DE-FC02-06ER25749 and #DE-FC02-06ER25755; NSF grants #CNS-0403342, #CCF-0702675 and #CCF-0833169; grants from Intel, Mellanox, Cisco systems, Linux Networx and Sun Microsystems;.

ing the potential for benefiting from the asynchronous nature of such operations.

In the recent past, the emergence of intelligent networking infrastructure (e.g., InfiniBand (IB) [16], Quadrics [20], Blue Gene[2]) has opened up new avenues allowing MPI implementations to alleviate these issues. Specifically, these networks provide native one-sided RDMA and one-sided atomic primitives that allow the MPI implementation to achieve true one-sided inter-node communication. At the same time, the rapid emergence of multi-core architectures in HEC has lead to large amounts of intra-node communication, thus requiring efficient support for true one-sided communication within the node as well. However, these two goals of efficient inter-node and intra-node communication are often contradicting—in order to use network-based one-sided communication, locking primitives provided by the network hardware need to be used and in order to use shared-memory-based one-sided communication, CPU atomic locking primitives need to be used. These two locks are distinct—a CPU cannot perform atomic operations on a network lock and vice-versa.

In our previous work, we utilized network one-sided and atomic operations allowing MPI to provide true one-sided communication. However, in that model, all one-sided communication, including that between processes on the same node, had to go over the network. This is clearly inefficient and is expected to add more performance overhead and network contention as the number of cores on each physical node grows. Thus, in this paper, we extend our previous work to natively take advantage of one-sided atomic operations on cache-coherent multi-core/multi-processor architectures while still utilizing the benefits of networks such as IB. Specifically, we first design fast locks within the node using CPU atomic operations and across nodes using IB hardware atomic operations. Then, we utilize these locks in a hybrid mechanism that allows dynamic migration between the CPU-based and network-based locks based on different policies. Our experimental evaluation shows that the hybrid design can overcome the limitations of the existing approaches (including our previous approach) and gives the best performance as well as potential for asynchronous communication.

The rest of the paper is organized as follows. In Section 2, we provide the background for our work. We describe some of our prior work in Section 3. We describe our new design in in Section 4. We evaluate our designs in section 5 and discuss the related work in section 6. Conclusions and future work are presented in section 7.

2 *Background*

In this section we describe MPI-2 one sided communication, details of IB and the issues in using atomic operations.

2.1 *One-sided Communication in MPI*

In MPI one-sided communication (also referred to as remote memory access or RMA), the origin process (the process that issues the RMA operation) can access a target process' remote address space directly. In this model, the origin process provides all the parameters needed for accessing the memory area on the target process (also referred to as *window*) using an *MPI_Put*, *MPI_Get* or *MPI_Accumulate* operation. The completion of these operations is guaranteed by explicit synchronization calls. One-sided semantics require that these memory accesses happen within an *access epoch* and an *exposure epoch* on the origin and target process, respectively. These epochs are the period between two synchronization calls on the origin and target processes. MPI provides two types of synchronization modes: (i) active synchronization, where both the origin and the target process make synchronization calls. (ii) passive synchronization, where only the origin process makes synchronization calls.

Passive synchronization is achieved through lock and unlock calls made only by the origin process. Such synchronization is convenient for applications utilizing these operations on large-scale systems due to its ability to minimize the coordination required between the origin and target processes, and increasing the potential for asynchronous communication. Thus, in this paper, we only concentrate on this form of synchronization.

2.2 *Overview of InfiniBand*

IB [16] is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. It provides one-sided RDMA communication primitives as well as RDMA atomic primitives. Remote Direct Memory Access (RDMA) [15] operations allow processes to access the remote process' memory without any intervention from the remote CPU.

RDMA Atomic Operations: IB provides two network level remote atomic operations, namely, *fetch_and_add* and *compare_and_swap*. The network interface card (NIC) on the remote node guarantees the atomicity of these operations. These operations act on 64-bit values. In the atomic *fetch_and_add* operation, the issuing process specifies the value that needs to be added and the remote address of the 64-bit location to which this value is to be added. In an atomic *compare_and_swap* operation, the issuing process specifies a 'compare value' and a 'new value'. The value at the remote location is atomically compared with the 'compare value' specified by the issuing process. If both the values are equal, the original remote value is swapped with the new value which is also provided by the issuing process. If these values are not the same, swapping does not take place. In both the cases, the original value is returned to the issuing process. In our design, the atomic compare and swap operations are used to implement efficient locking.

2.3 *Issues with coordinating network and shared-memory locks*

Most processor architectures provide fast atomic locks based on few CPU instructions. These can be used to implement locks efficiently across processes within the same node. As described above, networks such as IB provide network atomic operations that can be used to implement locks across nodes in an efficient and truly one-sided fashion. However, these two forms of locks are not interoperable. Specifically, network-based atomic operations achieve their atomicity through serialization at the network adapter. That is, the network adapter orders accesses to the atomic variable in the order in which it receives requests, thus guaranteeing that the variable is always in a consistent state. CPU-based atomic operations, on the other hand, do not pass through the network adapter at all, and are handled fully in the processor cache.

If both the CPU and the network try to work on the same lock, it is possible that the CPU fetches the variable to cache to perform an operation on it. At the same time, the network can trigger a cache flush through the chipset, forcing the variable to be in an inconsistent state.

In short, the CPU and the network need to work on different locks leading to several challenges in achieving lock coherence in a one-sided manner, that we will address in this work.

3 *Prior Work*

As described in the previous section the performance of lock/unlock operations is crucial to the performance of one-sided communication. One existing approach for lock/unlock is to use two-sided communication for implementing passive synchronization. In this approach the locking is implemented using a lock manager. As illustrated in Figure 1, every process has a lock manager to handle the incoming lock/unlock requests for its window. The lock manager queues the requests in a request queue and appropriately grants the lock. This results in target process involvement through the lock manager for every lock/unlock request leading to poor performance. In our previous work, we explored the use of IB atomic operations to design one-sided passive synchronization in MPI [21]. This work mainly targeted inter-node communication. In particular, the aim was to reduce the target involvement in one-sided communication. For every window on a target process we maintain a 64-bit global lock state that is registered with the network interface card (NIC) to support remote atomic operations.

This variable is initialized to the unlocked state during window creation. In order to acquire a lock, a network based atomic *compare-and-swap* operation is performed on this variable as shown in Figure 2. If the *compare-and-swap* operation is successful, the lock is obtained and the global state variable is set to the *process rank* of the origin process to indicate that it currently holds the lock. During the unlock operation, this value is set back to the default value. Other processes trying to acquire a lock at the same time would fail and would keep trying till the lock owner relinquishes the lock. There is no involvement of the target process.

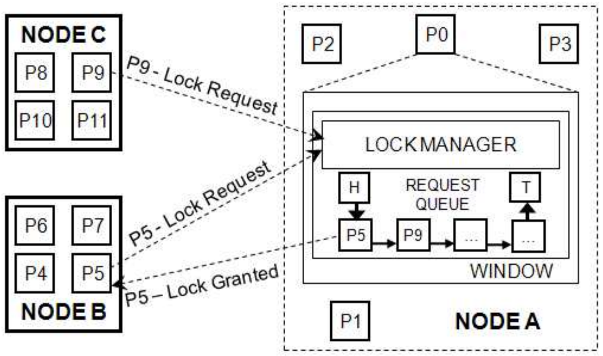

Figure 1: Two-sided Based locks

Another aspect that was achieved in this work was to highlight how the computation and communication overlap could be improved using truly one-sided passive synchronization. When two-sided approaches are used, the communication operations are often delayed to the synchronization phase and in some cases combined with an unlock synchronization call. In order to improve progress, which in turn leads to better overlap, the one-sided operations within the passive synchronization epoch are issued immediately using RDMA Write and RDMA Read operations. The completion of these operations is handled in the unlock operation. We demonstrated significant improvement using this approach for overlap on both the origin as well as the target side as compared to the two-sided based design. Henceforth, we will refer to this approach as ‘one-sided’ approach.

However, as mentioned earlier in this section, this work mainly targeted inter-node communication. With the increasing trend of more and more cores per node, intra-node communication assumes more significance. In this work, we address the issues of extending the previous design to handle both inter-node and intra-node communication efficiently.

4 *Migrating Locks for Multi-cores and High-speed Networks*

While using IB network atomic operations for one-sided communication allows for truly one-sided passive synchronization, this approach might not be the best in light of the increasing number of multi-core systems and the number of cores on each system. Specifically, using network operations to synchronize even between processes on the same node can

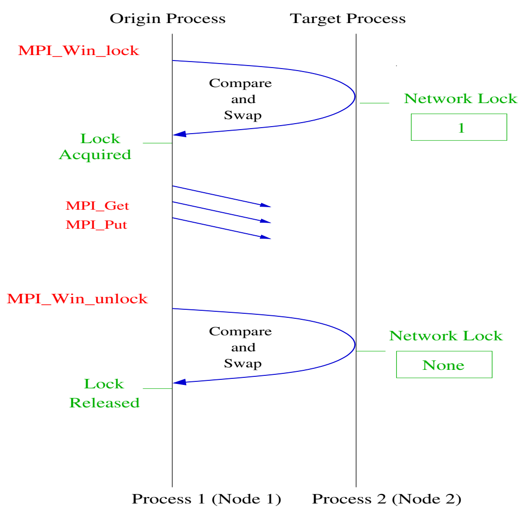

Figure 2: Network Based One-sided locks

have performance implications (since all the data has to traverse down to the network adapter and back) as well as network contention issues (since the network adapter is shared between all the cores). Thus, in this section, we propose a new hybrid design that utilizes CPU-based atomic operations in conjunction with network atomic operations to take advantage of both.

4.1 *Proposed Design*

Simultaneously utilizing both CPU-based atomic operations as well as network atomic operations is not trivial because of interoperability issues between these two operations as discussed in Section 2.3. Thus, there has to be a coordination mechanism between the network based locks and the CPU based locks. Our proposed solution to the problem is to migrate between the two locking mechanisms (network locks and CPU locks) when required. Since the locking is per-window based, different windows on the same process could be in a different locking mode depending upon the nature of the lock requests for that window.

Every node maintains the following state variables: (i) locking mode (network or CPU based), (ii) CPU lock and (iii) 64 bit global network lock. The locking mode variable and CPU lock variable are placed in shared memory so that other processes on that node can access it. The network lock can have the following values: (i) a value of 0 to (*MPI_Comm_size* - 1) indicates that the lock is in network mode and the actual value denotes the process that holds the network lock, (ii) a value of *MPI_Comm_size* indicates that it is unlocked, and (iii) a value of *MPI_Comm_size* + 1 indicates that the lock is in CPU mode.

In the network lock mode described in Figure 3, all the locks use IB atomic operations to obtain the network lock. In the CPU lock mode described in Figure 4, the intra-node locks use fast CPU based locks and the inter-node locks use a two-sided approach of sending the lock request to the lock manager (step 1) which then obtains the CPU lock on its behalf (step 2) and responds with lock granted (step 3).

By default, the lock is preset to one of the above two-modes, for example CPU based mode. When the mode needs to be migrated, a two-sided message is sent to the lock manager which acquires both the network as well as CPU lock, modifies the locking mode to ’network’, and then grants the lock. Any further locking now happens through IB atomic operations in a completely one-sided manner. The lock migration from a CPU mode to network mode is illustrated in Figure 5. When a remote process wants to acquire a lock, it performs a compare and swap with the network lock state (step 1). If the remote process discovers that the lock is in CPU mode, and it wants to migrate the lock to network mode, it sends a two-sided message to the lock manager requesting migration to network mode (step 2). The lock manager acquires both the network lock and the CPU lock (step 3), modifies the lock mechanism to CPU mode (step 4), and sends the lock granted packet to the remote process (step 5). A similar approach is done to reset the lock to CPU based. In this way, the locks can be migrated from one mechanism to other.

Thus, in summary, intra-node locks are completely one-sided as long as the lock is in CPU-mode and inter-node locks are completely one-sided as long as the lock is in network mode. If the lock is not in the appropriate mode, a two-sided synchronization is needed to migrate the lock to the appropriate mode. Henceforth we will refer to this approach as ‘Hybrid’.

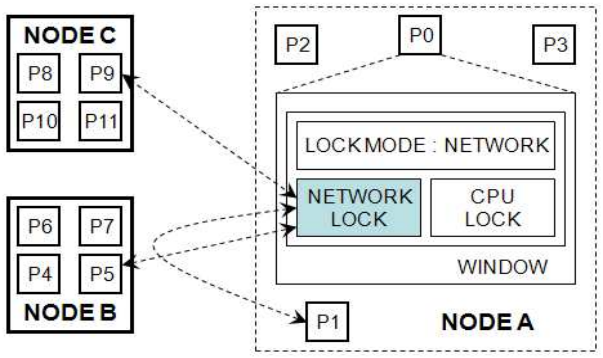

Figure 3: Locking Mechanisms: Network Lock

4.2 *Migration Policies*

Migration of locks could be based on various criteria. It could be based on (i) communication pattern, (ii) history, (iii) priority, (iv) native hardware capabilities and so on. The criteria used to migrate the locks is not the focus of this paper, and could be part of follow up work. In all the evaluations in this paper, the lock is preset to CPU mode for simplicity. Any remote node process lock request migrates the lock to network mode and any future intra-node lock request migrates the lock

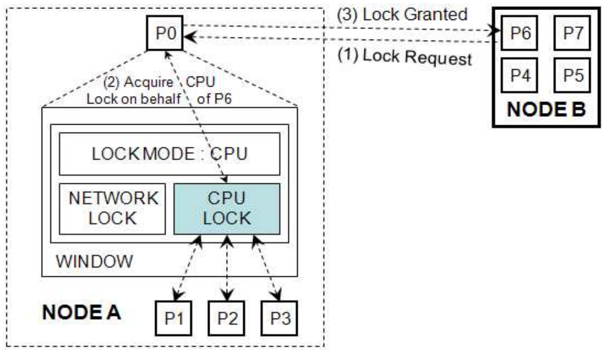

Figure 4: Locking Mechanisms: CPU Lock

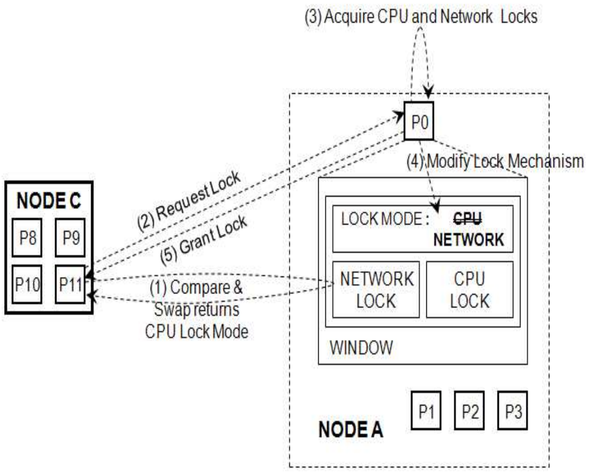
(3) Acquire CPU and Network Locks
P0
(4) Modify Lock Mechanism
LOCK MODE : ~~CPU~~ NETWORK
NETWORK LOCK
CPU LOCK
WINDOW
NODE A
P1
P2
P3
NODE C
P8
P9
P10
P11
(2) Request Lock
(5) Grant Lock
(1) Compare & Swap returns CPU Lock Mode

Figure 5: Locking Mechanisms: Lock Migration

to CPU mode.

5 *Experimental Results and Analysis*

In this section we evaluate the performance of our migrating locks based 'hybrid' design with the purely 'two-sided' based and the network based 'one-sided' approaches described in Section 3. We evaluate the performance for a wide range of scenarios. First, we evaluate and analyze the performance when the lock/unlock operations occur within the same node (intra-node) among the different cores. Then we show the performance when the operations are purely inter-node. Finally, we evaluate the performance for a combination of inter-node and intra-node operations. We also measure the overhead involved when the locks are migrated.

5.1 *Experimental Testbed*

Each node of our testbed has 16 AMD Opteron 1.95 GHz processors with 512 KB L2 cache. Each node also has 16 Gigabyte memory and PCI-Express bus. They are equipped with MT25418 HCAs with PCI-Ex interfaces. A 24-port Mellanox switch is used to connect all the nodes. The operating system used is RedHat Enterprise Linux Server 5.

5.2 *Intra-node Performance*

In this section, we first evaluate the performance of our new design for intra-node operations on a single node. Figure 6 shows the performance of lock/unlock operation comparing the three approaches. As expected our new hybrid design performs the best, since the lock/unlock operations within a node are basically few CPU instructions. In the two-sided approach, a lock request packet is sent to the lock manager of the target process. The lock manager responds with the lock granted packet. These lock requests and lock granted packets go over shared memory since the target is on the same node. In the one-sided based approach, the lock operation is achieved through an IB loop-back atomic fetch and add operation. Since the loop-back operation is expensive, it has the lowest performance for a single lock/unlock operation.

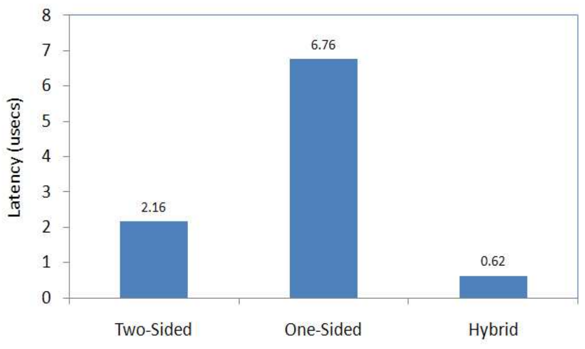

Figure 6: Lock/Unlock Performance

5.3 *Intra-node Performance with Remote Computation*

Next we evaluate the performance of the three approaches in the presence of computation on remote/target process. Minimal remote/target process involvement is important for one-sided passive synchronization calls so that the target can proceed with its computation. In this benchmark, the origin process acquires the lock and unlock operation on target process while computation is performed on the target process. The computation is a dummy loop that is executed on the remote/target process. In this experiment the performance of the three schemes is measured for varied amounts of dummy loop computation. The results are shown in Figure 7. Here the one-sided approaches (network based, one-sided and hybrid approach) is not affected with increasing amounts of computation on the target process, since it is not dependent on the target process to progress. Whereas, the performance of the two-sided scheme degrades with increasing amount of computation. This is expected because the two-sided approach requires target process involvement. In the presence of computation, it takes longer to respond to the lock/unlock requests.

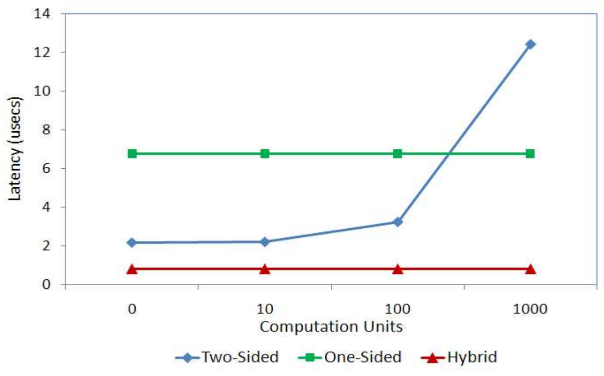

Figure 7: Lock/Unlock Performance with Remote Computation

5.4 *Concurrency and Contention*

Next we evaluate the performance of the different approaches when several lock/unlock operations occur concurrently. These experiments are conducted on a single node.

5.4.1 *Network Contention*

In the first micro-benchmark, each process locks its neighboring process (rank+1) on the same node. Thus in this benchmark, there are as many lock/unlock operations happening concurrently as the number of cores for which the benchmark is run. We measure the average latency of lock/unlock operation in this scenario. The results are shown in Figure 8. We observe that the two-sided performance is not degraded since the lock/unlock requests messages are sent over shared memory and there is no network contention. However the one-sided scheme using loop-back suffers degradation due to network contention since all the lock/unlock operations result in network transactions. In this scenario also, the hybrid scheme performs the best since the CPU based locks do not result in network contention.

5.4.2 *Lock Contention*

The next benchmark shows the performance of the three approaches when several processes are contending for a lock on the same window. The results are shown in Fig. 9. The hybrid scheme performs the best for up to three lock contentions. Beyond four contentions, the two-sided approach performs better than the hybrid scheme. The one-sided approach performs the least. This is expected since there would be lots of network transactions in the presence of contention.

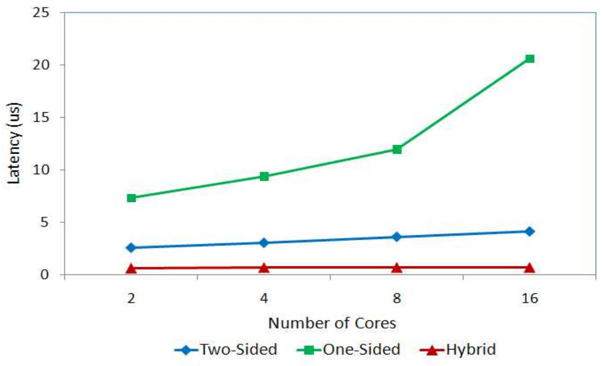

Figure 8: Lock/Unlock Performance with Network Contention

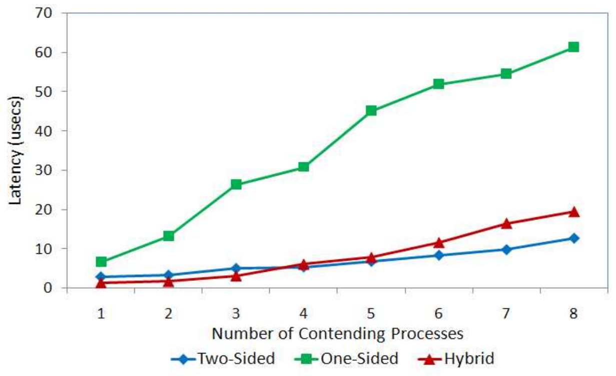

Figure 9: Lock/Unlock Performance with Lock Contention

5.5 *Inter-node Performance*

In this section, we compare the performance of the three approaches when the operations are purely inter-node. We use a micro-benchmark to demonstrate the benefits of one-sided approaches in the presence of computation and skew. We used Testbed B for this experiment, since we had more number of nodes to understand the inter-node performance. The experimental testbed (Testbed B) used for this benchmark is a 64 node Intel cluster. Each node of the testbed is a dual processor (2.33 GHz quad-core) system with 4GB main memory.

The benchmark simulates a ring type of communication wherein each process locks the window of its successor, puts some data in the target window and updates a tag indicating completion of the data transfer to that window. The target process then makes sure that the data is available in its window, then performs the same operation on its successor. The communication terminates when the message traverses through the complete ring. Simultaneously all the nodes are also performing computation in the form of a dummy loop. For the sake of simplicity, a fixed amount of computation is being performed by all the nodes. This benchmark evaluates the capability to overlap computation and communication. The results are shown in Figure 10. The one-sided and the hybrid approach outperforms the two-sided approach. This is due to the ability of the one-sided and hybrid approach to perform the lock/unlock operations in a truly one-sided fashion, whereas the two-sided approach requires remote host involvement to make progress. This results in delay for the target process in responding to lock requests. Since this benchmark is a ring type of communication, this could manifest itself as skew for the other processes further in the ring resulting in a cascading effect. In this scenario, the hybrid scheme remains in the network locking mode exclusively and hence its performance is similar to that of the one-sided approach.

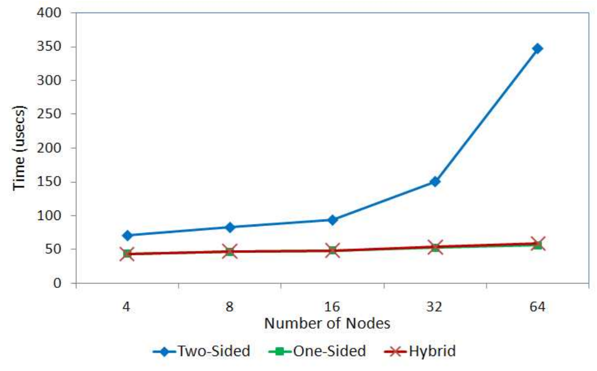

Figure 10: Inter-node Performance

5.6 *Lock Migration*

In this section, we try to evaluate the overhead incurred due to lock migration. The benchmark measures the average time taken for an intra-node lock/unlock operation and an inter-node lock/unlock operation in the presence of migration of the lock mechanism from network mode to CPU mode and vice-versa. The experiment is a two node experiment in which a process P1 acquires a lock/unlock on a process P0 on the same node 1000 times. During this duration, a process P2 on the second node tries to obtain the lock on P0 for x times triggering a migration each time.

The intra-node line in Figure 11 shows the latency of the lock/unlock operation happening on the same node with increasing percentage of migrations. We observe that for small percentage of migrations, the overhead is not very high as compared to case when no migrations occur. The inter-node line similarly shows the latency of the lock/unlock operation happening across nodes with increasing percentage of migrations. For smaller number of migrations, the overhead incurred is quite less. Large number of migrations lead to some overhead. However it is to be noted that, the biggest benefit achieved by this approach is to be able to maintain the truly one-sided nature of the locks once the migration has been achieved and thus provide greater potential for asynchronous communication as well as higher computation communication overlap. Also the migration policy described in Section 4.2 can be used appropriately to minimize the number of migrations.

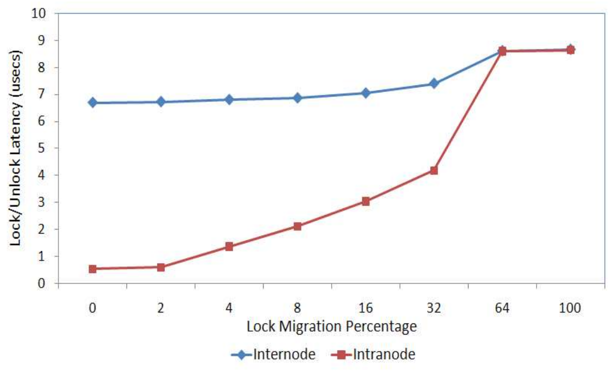

Figure 11: Lock Migration Overhead

5.7 *Hierarchical Task Sharing Communication Pattern Micro-benchmark*

Finally, we evaluate the performance for a combination of inter-node and intra-node operations with lock migrations by simulating a benchmark that performs task sharing and redistribution. The details of the benchmark is described below. The experiment is run on 4 nodes with 16 cores on each node for a maximum total of 64 cores. A hierarchy of leaders is created with one leader process designated on each node. First, the leader on every node performs 1000 Lock-Put-Unlock on every other local process on the same node. Then, the leader performs 1000 Lock-Put-Unlock on the leader of every other node. Finally, the leader on every node performs 1000 Lock-Put-Unlock on every local process again. The benchmark tries to simulate a scenario in which a leader process tries to get data/work from close neighbors, then gets data from remote neighbors in a cycle. The resulting communication pattern is a clique-based communication described in earlier sections. The results are shown in Figure 12. The communication pattern described above has lot more intra-node operations than inter-node operations. The hybrid scheme performs the best because it uses the fast CPU locks for the intra-node operations, and when the operations are inter node, it migrates to network mode. Thus it provides the best performance for such a communication scenario and we also observe that the performance gap is sustained for increasing number of processes.

6 *Related Work*

There are several studies regarding implementing one-sided communication in MPI-2. Some of the MPI-2 implementations that support one-sided communication are MPICH2 [6], OpenMPI [7],WMPI [17], NEC [22], SUN-MPI [9]. In OpenMPI, the library uses the two-sided approach for passive synchronization currently and depends on the target process making MPI calls to make progress. Besides MPI, there are

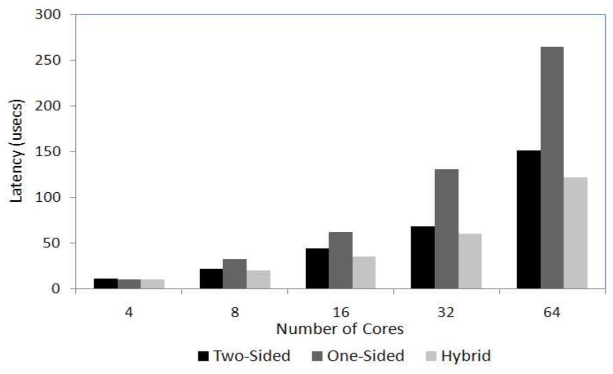

Figure 12: Hierarchical Task Sharing Communication Pattern

other programming models that use one-sided communication. ARMCI [19], GASNET [8] and BSP [13] are some examples of this model.

Researchers in [11] have proposed distributed queue based DLM using RDMA operations. Though this work exploits the benefits of RDMA operations for locking services, their design can only support exclusive mode locking. Researchers in [10] have studied efficient implementation of locks using NIC based atomic operations on Myrinet. Further, prior research in [18] extensively utilizes IB's remote atomic operations for shared and exclusive mode locking, however, the main focus in their work is not in the context of MPI-2 one-sided synchronization but rather as a system-wide distributed locking service typically used in data-centers.

7 *Conclusions and Future Work*

While MPI is the de-facto standard for communication in scientific applications, the usage of one-sided communication is restricted mainly owing to the inefficiencies in current MPI implementations. Specifically, current MPI implementations internally rely on synchronization between processes even during one-sided communication. This model is inherently susceptible to process skew, which consequently limits its ability to scale to large-scale systems. In this paper, we extended our previous work by proposing a hybrid model that dynamically migrates between hardware locks on networks such as IB and CPU-based atomic locks on multi-core architectures, to benefit from both. We presented our detailed design as well as experimental evaluation that showed significant performance improvement on a wide range of evaluations.

For future work we plan to extend our design to other networks such as the Blue Gene, and also evaluate the effectiveness of our design on various applications including the mpiBLAST bioinformatics application.

References

[1] Berkeley Unified Parallel C (UPC) Project. http://upc.lbl.gov/.

[2] Blue Gene System Architecture Overview. http://www.research.ibm.com/journal/rd/492/gara.html.

[3] Global Arrays. http://www.emsl.pnl.gov/docs/global/.

[4] GROMACS. http://www.gromacs.org/.

[5] PETSc. http://www-unix.mcs.anl.gov/petsc/.

[6] Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2/.

[7] B. W. Barrett, G. M. Shipman, and A. Lumsdaine. Analysis of Implementation Options for MPI-2 One-Sided. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[8] D. Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, Computer Science Division, University of California at Berkeley, October 2002.

[9] S. Booth and F. E. Mourao. Single Sided MPI Implementations for SUN MPI. In *Supercomputing*, 2000.

[10] D. Buntinas, D. K. Panda, and W. Gropp. NIC-Based Atomic Remote Memory Operations in Myrinet/GM. Workshop on Novel Uses of System Area Networks (SAN-1), February 2002.

[11] A. Devulapalli and P. Wyckoff. Distributed Queue Based Locking Using Advanced Network Features. In *ICPP*, 2005.

[12] M. Frigo and S.G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 2005.

[13] M. Goudreau, K. Lang, S. B. Rao, T. Suel, and T. Tsantilas. Portable and Effcient Parallel Computing Using the BSP Model. *IEEE Transactions on Computers*, pages 670–689, 1999.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[15] J. Hilland, P. Culley, J. Pinkerton, and R. Recio. RDMA Protocol Verbs Specification (Version 1.0). Technical report, RDMA Consortium, April 2003.

[16] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.

[17] F. E. Mourao and J. G. Silva. Implementing MPI's One-Sided Communications for WMPI. In *EuroPVM/MPI*, September 1999.

[18] S. Narravula, A. Mamidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations. CCGrid, 2007.

[19] 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.

[20] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The quadrics network (qsnet): High-performance clustering technology. In *In HotI 01*, pages 125–130, 2001.

[21] G. Santhanaraman, S. Narravula, and D. K. Panda. Designing Passive Synchronization for MPI-2 One-Sided Communication to Maximize Overlap. In *IPDPS*, 2008.

[22] J. Traff, H. Ritzdorf, and R. Hempel. The Implementation of MPI-2 One-Sided Communication for the NEC SX. In *Proceedings of Supercomputing*, 2000.