Optimizing Synchronization Operations for Remote Memory Communication Systems[*]

Darius Buntinas[1] Amina Saify[2] Dhabaleswar K. Panda[1] Jarek Nieplocha[3]

[1]Network-Based Computing Laboratory
Dept. of Computer and Information Science
The Ohio State University
{buntinas, panda}@cis.ohio-state.edu

[2]Scalable Systems
Dell Computer Corporation
amina_saify@dell.com

[3]Computational Sciences & Mathematics
Pacific Northwest National Laboratory
j_nieplocha@pnl.gov

Abstract

Synchronization operations, such as fence and locking, are used in many parallel operations accessing shared memory. However, a process which is blocked waiting for a fence operation to complete, or for a lock to be acquired, cannot perform useful computation. It is therefore critical that these operations be implemented as efficiently as possible to reduce the time a process waits idle. These operations also impact the scalability of the overall system. As system sizes get larger, the number of processes potentially requesting a lock increases. In this paper we describe the design and implementation of an optimized operation which combines a global fence operation and a barrier synchronization operation. We also describe our implementation of an optimized lock algorithm. The optimizations have been incorporated into the ARMCI communication library. The global fence and barrier operation gives a factor of improvement of up to 9 over the current implementation in a 16 node system, while the optimized lock implementation gives up to 1.25 factor of improvement. These optimizations allow for more efficient and scalable applications.

1. Introduction

Aggregate Remote Memory Copy Interface (ARMCI) [13] is a portable one-sided communication library compatible with message-passing libraries such as MPI [11] or PVM [19]. It has been used to for implementing distributed array libraries, such as Global Arrays [14], other communication libraries, such as Generalized Portable SHMEM [17], and compiler run-time systems such as PCRC Adlib [6] or more recently Rice Co-Array Fortran compiler (http://www.pmodels.org). ARMCI provides remote memory copy, accumulate and synchronization operations optimized for non-contiguous data transfers. Among the synchronization operations are *fence* operations, which ensure that previous memory operations have completed, as well as lock operations.

Such synchronization operations are necessary for many parallel operations accessing shared memory. However, a process which blocked waiting for a fence operation to complete, or for a lock to be acquired, cannot perform useful computation. It is therefore critical that these operations be implemented as efficiently as possible to reduce the time process wait idle. These operations also impact the scalability of the overall system. As system sizes get larger, the number of processes potentially requesting a lock increases. Similarly, the number of potential remote memory operations increases, increasing the number of nodes which must be contacted for a fence operation increases. So efficient implementation of these operations is also critical for scalability of the system.

In this paper we have described our design and implementation of a new ARMCI operation which combines a global fence operation with a barrier synchronization operation. We have also described how we incorporated an improved algorithm for locking local and remote locks in ARMCI. We then evaluate the performance of these improvements comparing them to the original implementation.

The combined operation of the global fence and barrier synchronization is used in the Global Arrays `GA_Sync()` function. The original implementation performed a global fence by having every process contact each remote process to ensure that all memory operations it issued had completed. The running time of this algorithm is linear in the number of processes in the system. We propose a new logarithmic time op-

[*]This research is supported in part by a DOE grant #DE-FC02-01ER25506 and NSF Grants #EIA-9986052 and #CCR-0204429.

[2]A portion of this work was performed as part of this author's masters thesis while she was a graduate student at the Ohio State University

eration, `ARMCI_Barrier()`, that performs the global fence operation and performs a barrier synchronization operation. Our new implementation gives a factor of improvement of up to 9 over the original implementation in a 16 node system.

The original lock implementation is a hybrid algorithm which uses a ticket-based lock algorithm for locking local locks and a server-based algorithm for locking remote locks. We implemented a software queuing lock algorithm [10] using atomic memory operations which reduced the lock synchronization time, and improved overall lock performance. We have observed up to a 1.25 factor of improvement over the original implementation in a 16 node system.

The rest of the paper is organized as follows. We will briefly describe the ARMCI library in the next section. In Section 3 we will describe the design and implementation of the `ARMCI_Barrier()` and lock operations. We evaluate the performance of our modifications in Section 4, and present our conclusions and future work in Section 5.

2. ARMCI - Aggregate Remote Memory Copy Interface

ARMCI is a portable communication library that offers remote memory copy functionality. ARMCI offers both a simpler and lower-level model of one-sided communication than MPI-2 which improves its performance [16, 7]. The ARMCI specification does not describe or assume any particular model for the lower-level implementation, for example threads or active messages.

In scientific computing, applications require transfer of non-contiguous data. With remote copy APIs which support only contiguous data transfer, it is necessary to transfer non-contiguous data using multiple communication operations. ARMCI, however, is optimized for non-contiguous data transfer. It is meant to be used primarily by library implementors rather than application developers. Examples of libraries that ARMCI is aimed at include Global Arrays, P++/Overture and PCRC Adlib run time system.

Applications frequently use a hybrid programming model, requiring both shared memory and message passing operations. For this reason, ARMCI is designed to be compatible with several separate message passing libraries, such as MPI and PVM.

It is very important for a communication library such as ARMCI to have straightforward progress rules. Simple progress rules simplify the development and performance analysis built on top of libraries that use ARMCI, and avoid dealing with ambiguities of the platform-specific implementations. Therefore, the ARMCI remote copy operations are truly one sided, and complete regardless of the actions taken by the remote process.

To support the full set of remote memory operations on a cluster of workstations using GM[12], ARMCI uses a client-server architecture [15]. Figure 1 shows the client architecture on a cluster of SMP nodes. Each node has a server thread which handles remote memory operations for each of the user processes running on the node. When a user process wants to perform a memory operation on the remote memory of a process, it sends a

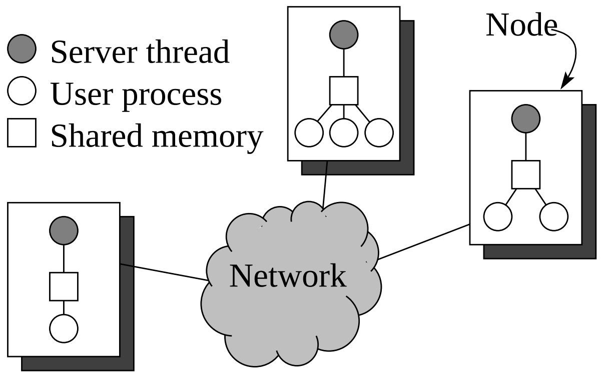


Figure 1. Client-server Architecture of ARMCI

request to the server thread at the node where the remote process is running. Each user process shares a memory region with the server thread, so when the server thread receives a request, it performs the operation on the memory region for that process. In order to reduce the processor usage by the server thread when the server is idle, the server will use blocking receives and sleep while waiting for incoming requests.

ARMCI supports data transfer operations such as *get* and *put*, atomic operations such as *accumulate* and *read-modify-write*, distributed mutex operations such as *lock* and *unlock*, as well as progress and ordering operations such as *fence* and *allfence*.

ARMCI allows user processes to issue non-blocking put operations on remote memory. For example, a process can issue a put operation to write a value in remote memory, but does not have to wait until the operation has completed. A put operation is considered complete when the value is actually written to the destination location. When a process issues a put operation on remote memory, a message is sent to the server at the remote node containing the data to be written and the location of where it should be written. The server receives the message and writes the data in the appropriate location. Because the put operation is non-blocking, the process does not have to wait for the message to be received, and for the operation to be completed, improving the performance of the system.

A side-effect of using non-blocking put operations is that the process does not know when a put operation completes. A special operation, called fence, is provided to guarantee that all previous put operations have completed. A call to the `ARMCI_Fence()` function returns when all put operations previously issued to a specified server have completed. ARMCI also provides a function called `ARMCI_AllFence()` which returns when all previously issued put operations have completed on all servers.

3. Design and implementation

In this section we describe our modifications of ARMCI. We first present the `ARMCI_Barrier()` function which combines a global fence operation with a barrier synchronization. We then describe how we incorporated software queuing locks into ARMCI.

```
x = N / 2;
while(x > 0) {
  send op_init[0..N-1] to process (my_id XOR x);
  receive into temp[0..N-1] from process (my_id XOR x);
  op_init[0..N-1] = op_init[0..N-1] + temp[0..N-1];
  x = x / 2
}
```

Figure 2. Algorithm for distributing `op_init[ ]` elements over N processes, where N is a power of two

3.1. ARMCI barrier

The `ARMCI_Barrier()` function combines two operations which are commonly used together in a single, more efficient function. These functions are `ARMCI_AllFence()` and a barrier synchronization function provided by the message passing library. ARMCI is designed to work with either the MPI or PVM message passing libraries. In this paper we describe the implementation using MPI. However, an implementation over PVM would be similar. The `ARMCI_AllFence()` function is used to ensure that all put operations have completed at remote nodes. The `MPI_Barrier()` function is a *barrier synchronization* operation which ensures that all processes have reached the same point in the code. The `ARMCI_Barrier()` function will ensure that all put operations to remote nodes have completed and also perform a barrier synchronization operation.

3.1.1 Existing implementation of `ARMCI_Fence()`

Depending on the underlying communication subsystem, ARMCI will use two different algorithms to perform an `ARMCI_Fence()` function. In certain communication subsystems, such as LAPI or VIA, put messages generate acknowledgement messages from the server for flow control, and reliability. When the server thread has completed the put operation, it sends an acknowledgement back to the initiating process. When `ARMCI_Fence()` is called, the process simply has to wait until it receives acknowledgements for all put requests sent to that server. However, with other communications subsystems, such as GM, there is no need for the server thread to send acknowledgements for put operations. In this case the `ARMCI_Fence()` function must send a message to the server requesting a confirmation that all put operations have been completed. The server sends this confirmation once it has done so.

In communication subsystems in which acknowledgement messages are not sent for put operations, the `ARMCI_AllFence()` function sends requests to *each* server requesting confirmation. The communication time a process spends to perform this operation can be as high as $2(N-1)$ one-way message latencies, where N is the number of processes.

We designed the `ARMCI_Barrier()` function to be used as a replacement for the `ARMCI_AllFence()` in the case when all processes are performing the `ARMCI_AllFence()` function concurrently, for example, in the Global Arrays `GA_Sync()` function. Our new function is designed to significanly reduce the communication time.

3.1.2 Our design

The `MPI_Barrier()` function is performed using a binary-exchange communication pattern. The operation proceeds in $\log_2(N)$ phases. In a phase, a process, a, sends a message to another process, b, and waits for a message from process b, before going on to the next phase. Because the messages in a phase can be overlapped, the communication time of this operation is $\log_2(N)$ one-way message latencies. So performing an `ARMCI_AllFence()` followed by a `MPI_Barrier()` would take $2(N-1)+\log 2(N)$ one-way latencies.

The new `ARMCI_Barrier()` is semantically equivalent to performing an `ARMCI_AllFence()` function followed by a `MPI_Barrier()` function. The `ARMCI_Barrier()` function is performed in three stages. First, each user process determines how many put requests were sent to its server thread. Next, each user process waits until all of the requests have completed at its server thread. Finally, the processes perform a barrier synchronization. The barrier synchronization ensures that no process can continue until all put requests have been completed at all server threads.

In our implementation each user process keeps track of the number of put operations it issues to each server thread, in an array `op_init[ ]`. Each server thread keeps track of the number of put operations it has completed, in a variable `op_done`. In the first stage, the `op_init[ ]` arrays are distributed among the user processes so that each process knows how many put requests were sent to its server. Such an operation is called an *all-scatter* or an *all-to-all* operation. We implemented this operation using a binary-exchange algorithm. Figure 2 shows the algorithm when the number of processes is a power of 2. This algorithm proceeds in $\log_2(N)$ phases. In each phase the process will exchange messages with another process. Since the messages in a phase can be overlapped, the communication time for this algorithm is $\log_2(N)$ message latencies.

At the end of the first stage of the `ARMCI_Barrier()` function, the value of the ith element of the `op_init[ ]` array at process i is equal to the number of put requests sent to the server thread of process i by all processes in the system. In the second stage, each process will wait until the value of its `op_done` variable is equal to the value of its element of the `op_init[ ]` array. The server thread of a process will increment the `op_done` variable as it completes incoming send requests. Once the server thread has completed all of the put requests sent to it, the value of `op_done` will match the corresponding value of `op_init[ ]`, and the process will proceed to the third stage, which is to perform a barrier synchronization operation. A binary-

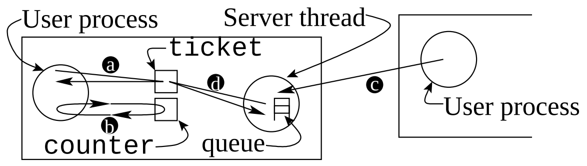


Figure 3. Local and remote lock operations

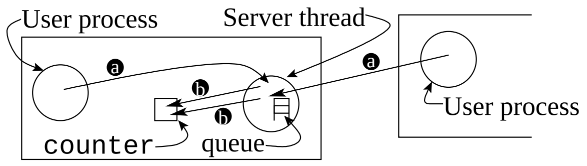


Figure 4. Local and remote unlock operations

exchange algorithm similar to the one used in the first stage is used for the barrier synchronization operation, so the communication time of this operation is $\log_2(N)$ message latencies. The total communication time of the `ARMCI_Barrier()` function is $2\log_2(N)$ message latencies which is considerably smaller than the communication time for performing an `ARMCI_AllFence()` followed by an `MPI_Barrier()`.

Note that in certain situations, such as when processes perform put operations on memory locations at less than $\log_s(N)/2$ other processes, the original implementation may provide better performance. In such a case the communication time to contact the servers to which put operations have been sent, will be less than the time to perform the binary-exchange algorithm to exchange the `op_init[ ]` arrays. An alternative implementation would be to allow the programmer to choose which algorithm to use in the `ARMCI_Barrier` function when the communication pattern of an application is known.

3.2. ARMCI lock

Currently, ARMCI uses a hybrid locking mechanism [16], where local locks are locked using a *ticket-based* algorithm, while remote locks use a *server-based queue* algorithm. This method requires the use of the server thread even if the lock is local, and the next process to acquire the lock is located on the same node. There are many algorithms for implementing locking on parallel or distributed systems, such as QOLB [8], LH and M [9], Raymonds [18], and Naimi Trehel [20]. We propose using a *software queuing lock* algorithm based on the MCS lock [10] which eliminates unnecessary involvement of the server thread, and reduces the lock synchronization time. In this section we will describe the exisiting algorithm, then describe our proposed algorithm and its implementation.

3.2.1 Existing lock algorithm

The existing ARMCI lock algorithm is a hybid algorithm combining ticket-based locking and server-based queue locking. Because ticked-based locks require polling on a variable, they are not well suited for remote locks. Server-based locks require interaction with the server thread which can be reduced when the lock is local, allowing processes to directly access the lock variables. Combining the two methods gives a more general algorithm.

In ticket-based locking, a lock consists of two variables, `ticket` and `counter`, located at a process. These are both initialized to zero. To request a lock, a process gets a unique ticket number by performing an atomic *fetch-and-increment* operation on `ticket`. The process is now guaranteed to have a unique ticket number. The process now polls on the `counter` variable until its ticket number is equal to the value of `counter`. When these two values are equal, the process has acquired the lock. To release the lock the process increments its ticket number and writes this value to the `counter` variable, thus allowing the next process waiting on the lock to acquire it.

In the hybrid algorithm, when a process requests a lock located on a remote process, it sends a lock request message to the server at that node, and waits for a reply. Upon receiving a lock request, the server thread takes a ticket number on behalf of that process by performing an atomic *fetch-and-increment* operation on `ticket`. If this ticket number is equal to the value of `counter`, then the server sends a reply message to the process, notifying the process that it has acquired the lock. If the ticket number is not equal to the value of `counter` the server puts the process' request on a queue until the values are equal. Processes requesting local locks follow the same steps for locking described above for ticket-based locks.

Figure 3 gives an example of a user process requesting a local lock and a user process requesting a remote lock. The user process on the left requests a local lock by a) performing an atomic *fetch-and-incrment* operation on the `ticket` variable, then b) polling on the `counter` variable. The user process on the right requests a remote lock by c) sending a lock request to the server thread at the node where the lock is located. The server thread then d) performs an atomic *fetch-and-incrment* operation on the `ticket` variable on behalf of the remote process and queues the request until the processes ticket number is equal to the value of the `counter` variable.

When a process unlocks a lock, whether it is local or remote, it contacts the server which increments the `counter` variable, then checks if the request at the head of the queue associated with the lock has acquired the lock, and sends that process a reply if so.

Figure 4 gives an example of a user process unlocking a local lock and a user process unlocking a remote lock. The user process on the left is unlocking a local lock and the user process on the right is unlocking a remote lock. Both processes unlock the lock in the same manner, by a) contacting the server thread at the node where the lock is located. The server thread, then b) increments the `counter` variable and checks if any queued lock requests have acquired the lock.

Notice that the existing lock mechanism requires that the server thread be contacted whenever a lock is released, even if the lock is local to the process releasing the lock. This impacts the time to pass the lock from

```
struct node {
  struct node *next;
  int locked;
} *mynode, *prev_node;
struct node *Lock;

request (Lock, mynode) {
  mynode->next = NULL;
  prev_node = swap (Lock, mynode);
  if (prev_node != NULL) {
    mynode->locked = TRUE;
    prev_node->next = mynode;
    while (mynode->locked) {};
  }
}
release (Lock, mynode) {
  if (mynode->next == NULL) {
    if (compare&swap (Lock, mynode, NULL))
      return;
    while (mynode->next == NULL) {};
  }
  mynode->next->locked = FALSE;
}
```

Figure 5. Pseudocode for software queuing locks

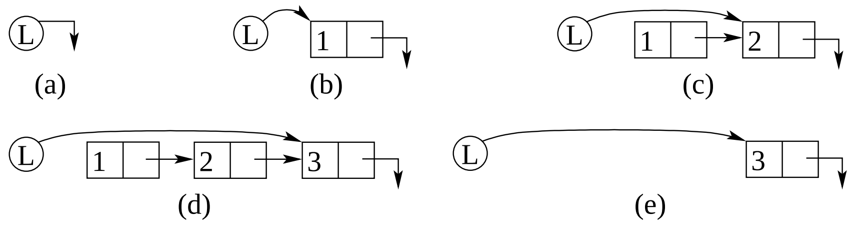


Figure 6. Example of locking and unlocking with software queuing locks

one process to the next waiting process, because of the communication time to send the lock release request, as well as the time to wake the sleeping server thread. Furthermore, if the next process waiting for the lock is not on the same node as the lock, then the server thread will have to send a message to that process indicating that it has acquired the lock. So up to two messages are required to pass the lock. In the next section we present our implementation of software queuing locks. Software queuing locks promise to eliminate the shortcomings of the existing implementation. Using this method, locks can be passed using only one message, or even zero messages, if the next waiting process is on the same node as the process holding the lock. This method also eliminates the need to involve the server when the processes requesting the lock, and the lock itself, are all on the same node.

3.2.2 Our design

We propose to use a software queuing lock to improve locking performance in ARMCI. First we will describe the general algorithm, and describe how we have implemented it. Figure 5 shows the pseudocode for the software queuing lock. In this algorithm, each process has a data structure, a `node` structure, consisting of a `next` pointer and a `locked` variable. A lock consists of a `Lock` variable located in global memory. The algorithm works by constructing a linked list of processes requesting a lock, with the process at the head of the list holding the lock. Initially, the `Lock` variable is set to `NULL`.

When a process requests the lock it first sets its `next` variable to `NULL` then performs an atomic *swap* operation swapping the value in `Lock` variable with a pointer to its `node` structure. If the value stored in the `Lock` variable was `NULL`, then the process has acquired the lock. If the `Lock` variable was not `NULL`, then the `Lock` variable held a pointer to the `node` structure of the last process waiting on the lock. In this case, the process adds itself to the linked list by first setting its `locked` variable to `TRUE` then setting the `next` pointer of the last process waiting on the lock to point to the its `node` structure. It then polls on its `locked` variable, waiting for `locked` to become false.

When a process releases a lock, it checks if another process is waiting for the lock, by checking the `next` variable. If this is not `NULL` it sets the `locked` variable of the `node` structure pointed to by the `next` variable to `FALSE` allowing the next process waiting on the lock to acquire it. If, however, the `next` variable is `NULL` the process must check whether another process is in the process of requesting the lock and has not yet set this process' `next` variable. To do this it does this by performing an atomic *compare&swap* operation on the `Lock` variable, setting the `Lock` variable to `NULL` only if no other process has requested the lock and the `Lock` variable still points to this process' `node` structure. If no other process has requested the lock, then this process has released the lock. If, however, the `Lock` variable was not pointing to the `node` structure of this process, then another process has updated the `Lock` variable, but has not yet written to this process' `next` pointer. In this case, the process will wait until that process writes to the `next` variable. It will then set the process' `locked` variable to `FALSE` allowing that process to acquire the lock.

Figure 6 shows an example of locking and unlocking using software queuing locks. We see in a) a free lock where the `Lock` variable is set to `NULL`. In b) we see that Process 1 has acquired the lock, and that no other processes have requested the lock, so its `next` pointer is set to `NULL`. In c) Process 2 has requested the lock, but since Process 1 still holds the lock, it is added to the queue. Notice that the `next` variable of Process 1 points to the `node` structure of Process 2, and since Process 2 is the last process waiting on the queue, the `Lock` variable also points to the `node` structure of Process 2. We see in d) that Process 3 has requested the lock, so the `next` pointer of Process 2, as well as the `Lock` variable, now point to the `node` structure of Process 3. In e) we see that Process 1 has released the lock, allowing Process 2 to acquire and then release the lock, leaving Process 3 holding the lock. After Process 3 releases the lock, the `Lock` variable will be set to `NULL` like it was in a).

In ARMCI, remote memory is referenced using a tuple of the remote process' id number and the virtual memory address at the remote process. This means that in order to implement the software queuing lock on ARMCI, the `next` and `Lock` pointers must be represented as such tuples. However the atomic memory operations in ARMCI only support *integer* or *long* operands. In order to implement the software queuing locks, we added new atomic memory operations which operate on pairs of *long* variables. Since ARMCI did not have an atomic *compare&swap* operation we also added this function.

To implement the locks, each process allocates a global `Lock` variable for each lock that will be located at that process. For example, if three locks are to be created one at Process 1, another at Process 4 and the third at Process 11, each of these processes would allocate one `Lock` variable. The other processes need only have pointers to these variables. Each process must also allocate its `node` structure. Note that only one `node` structure is needed per process regardless of how many `Lock` variables are allocated. To request or release a lock, a process performs the procedures described above, using the new atomic memory operations to when operating on remote memory pointers. The statements on lines 9, 12, 18 and 22 of Figure 5 are the statements which access another process' memory. In our implementation, we used appropriate atomic operations or put operations in those statements.

In ARMCI when performing get, put and atomic memory operations the user process checks to see if the referenced memory is local or remote. If the memory is local, it performs the operation directly. If the memory is remote, it contacts the server thread at the remote node to perform the operation. This means that, with software queuing locks, when a process is unlocking a local lock the server thread is not involved if there is no other process waiting for the lock. Furthermore, when requesting a local lock, the server thread will not be involved unless the process waiting for the lock immediately before this process is remote, and when releasing a local lock, the server thread will not be involved unless the process waiting for the lock immediately after this process is remote.

The existing hybrid approach ARMCI uses, requires that the process contacts the server each time it releases the lock. Also the *lock synchronization time*, i.e., the time between when one process releases the lock and the next waiting process acquires the lock, for remote locks in the existing approach is two message latencies: The process releasing the lock sends a message to the server thread, which then sends a message to the next waiting process. In software queuing locks, the process releasing the lock directly contacts the next waiting process, so the synchronization time is one message latency. However, when releasing a lock when there is no other process waiting for it, the software queuing lock performs an atomic *compare&swap* operation. For remote locks, this means that the process must contact the the server at a remote node, and then wait for a response. The existing algorithm does not have to wait for a response from the server. This means that the time a process spends releasing the lock will be higher for the software queuing locks in such a case, however we expect that the benefits described above will improve overall system performance despite this case.

4. Performance evaluation

In this section we evaluate our modifications against the original implementation. We performed our evaluation on a 16 node cluster consisting of 1GHz dual-SMP Pentium III nodes with 32MHz/32 bit PCI slots. The cluster was connected using a Myrinet-2000 network.

4.1. ARMCI barrier evaluation

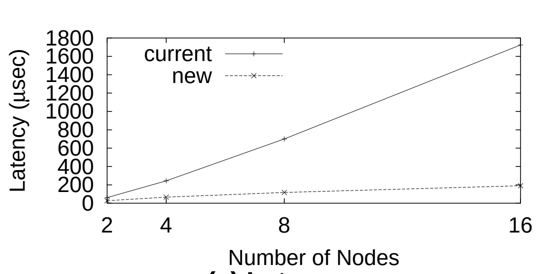
To evaluate our `ARMCI_Barrier()` function, we modified the Global Arrays library to use this function in the `GA_Sync()` function. We then evaluated the performance of this function and compared it with the original implementation. In our test, we created a two dimensional array which is distributed uniformly over the set of processes, and had each process write values into portions of the array which are remote to them. Next, we performed an `MPI_Barrier()` operation to make sure that all of the processes have completed this step, then we called `GA_Sync()` and timed it. We performed this test 100 times and took the average time for all iterations over all processes.

By writing to the array, we ensured that the processes would have to perform fence operations with each other in `GA_Sync()`. We called `MPI_Barrier()` before calling `GA_Sync()` to synchronize the processes in order to ensure that the times we were reporting were not due to process skew.

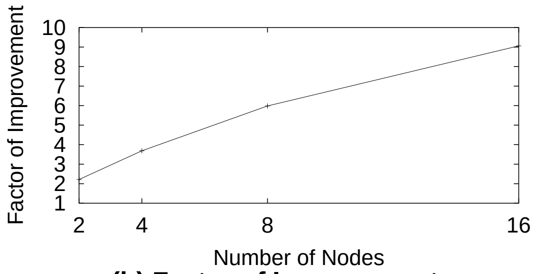
Figure 7 shows the results of this test. In Figure 7(a) we see that the new implementation performs considerably better than the current. The new implementation can perform the `GA_Sync()` operation in 190.3μs while the current implementation takes 1724.3μs for 16 processes. Figure 7(b) shows the factor of improvement for the new implementation over the current implementation. We see a factor of improvement of up to 9 for 16 processes.

4.2. ARMCI lock evaluation

In order to test our new lock implementation, we had each node repeatedly request and release a lock located at one of the processes. We then timed how long each of these operations took. We performed 10,000 iterations of this test and took the average times over all iterations and over all processes. By varying the number of processes we varied the load on the lock. When only one process is performing the test, we took two cases, one where the lock was local and one where the lock was remote. The numbers which we reported in the graphs are a average of these two.

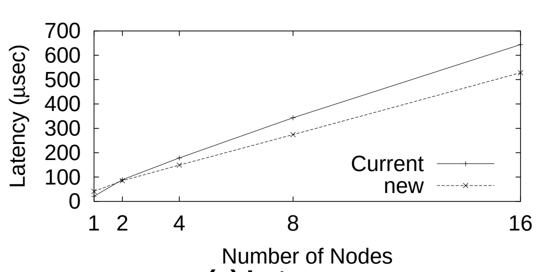


(a) Latency

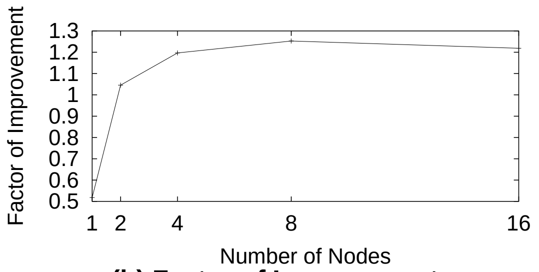


(b) Factor of Improvement

Figure 7. Comparison of current implementation of `GA_Sync()` and new implementation of `GA_Sync()` which uses the new `ARMCI_Barrier()` function



(a) Latency



(b) Factor of Improvement

Figure 8. Time to request, acquire and release a lock

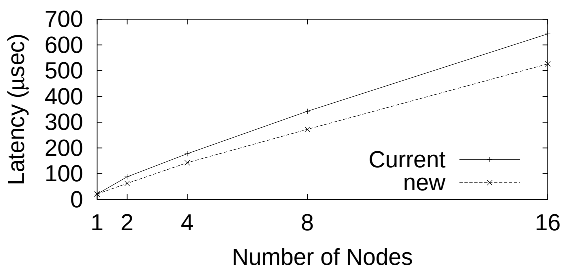


Figure 9. Time to request and acquire a lock

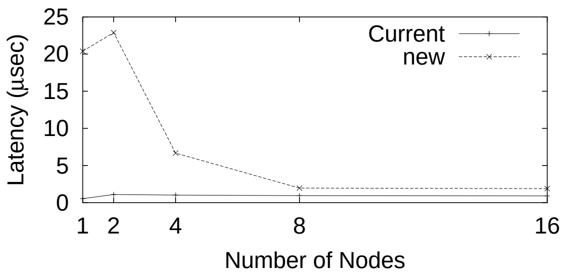


Figure 10. Time to release a lock

Figure 8 shows the results of this test. Figure 8(a) shows the average time each process spends requesting and releasing the lock. Notice that when two or more processes are competing for the lock, the new lock implementation outperforms the current one. This is because in the new implementation passing the lock from one process to the next waiting process only requires one message, while the current implementation requires two messages. However, when there is no process waiting for the lock in the new implementation the process performs a *compare&swap* operation which means that the process must send a message to the server and wait for a reply. In the current implementation, the process simply has to initiate sending a message to the server and need not wait for a reply. We see this in the results for the one process case. In this case, every time the process releases the lock, it must perform a *compare&swap* operation, so the results for the new implementation are higher than for the current. Figure 8(b) shows the factor of improvement for this test. We see up to a 1.25 factor of improvement of the new implementation over the current for the eight node case. Note that although the factor of improvement decreases slightly for 16 nodes, the new implementation continues to outperform the current implementation. In fact, the time taken by the current implementation continues to grow faster, as the number of nodes increases, than that for the new implementation.

Figure 9 shows just the time to request and acquire the lock. We see that the new implementation always outperforms the current one. As mentioned previously this is because of the reduced time to pass the lock from one process to the next. Figure 10 shows just the time to release a lock. Here we see that the new implementation takes more time than the current. This is because of the *compare&swap* operation that is performed. This operation is only performed when there is no other process waiting for the lock. As the number of nodes increases, the chance that there is no other process waiting for the lock decreases, so the average decreases. Even though the time to release a lock is higher in the new implementation, this is offset for all but the single process case by the lower time to request and acquire the lock, so the overall performance of locking in ARMCI is improved. We are currently investigating optimizations which would eliminate the need for a *compare&swap* operation when releasing a lock. Such an optimization would improve the performance of unlocking a lock when there is no other process waiting on that lock.

5. Conclusions and future work

In this paper we have described the design and implementation of optimized synchronization operations. These operations gave a significant improvement over the current implementation, specifically, our `ARMCI_Barrier()` operation gave a factor of improvement of up to 9 when used in the Global Arrays library, and our optimized lock implementation gave up to a 1.25 factor of improvement over the current implementation. Such improvements will help improve the scalability of applications and systems that use such synchronization operations.

In future work we intend to investigate methods of reducing or eliminating the interaction between user processes and the server thread. This can be performed through the use of NIC-based operations [1, 2, 4, 5, 3]. We are currently investigating which NIC-based operations would provide the best benefit to the application. More immediately, we are working on optimizing the lock operation to eliminate the need for the *compare&swap* operation when releasing a lock.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy at Ohio State University and Pacific Northwest National Laboratory. PNNL is operated for DOE by Battelle Memorial Institute. This work was supported by the Center for Programming Models for Scalable Parallel Computing sponsored by the MICS Division of DOE's Office of Computational and Technology Research.

Additional Information

Additional papers related to this research can be obtained from the following Web pages: Network-Based Computing Laboratory (http://nowlab.cis.ohio-state.edu), Parallel Architecture and Communication Group (http://www.cis.ohio-state.edu/∼panda/pac.html) and the ARMCI webpage (http://www.emsl.pnl.gov:2080/docs/parsoft/armci).

References

[1] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Proceedings of the 27th International Conference on Parallel Processing (ICPP '98)*, pages 381–390, August 1998.

[2] D. Buntinas, D. Panda, and W. Gropp. NIC-based atomic remote memory operations in Myrinet/GM. In *Workshop on Nove Uses of System Area Networks (SAN-1)*, February 2002.

[3] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan. Broadcast/Multicast over Myrinet using NIC-Assisted Multidestination Messages. In *Proceedings of Int'l Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC)*, pages 115–129, 2000.

[4] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001, (IPDPS)*, April 2001.

[5] D. Buntinas, D. K. Panda, and P. Sadayappan. Performance Evaluation of NIC-level Barrier over Myrinet/GM. In *Proceedings of Int'l Workshop on Communication Architecture for Clusters (CAC)*, 2001.

[6] B. Carpenter, G. Zhang, and Y. Wen. NPAC PCRC runtime kernel definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997. Up-to-date version maintained at http://grids.ucs.indiana.edu/ptliupages/projects/HPJava/pcrc/docs.html.

[7] C. S. Guiang, A. Purkayastha, and K. F. Milfeld. Remote memory operations on Linux clusters using the Global Arrays tooklit, GPSHMEM and ARMCI. In *Linux Clusters: The HPC Revolution*, October 2002.

[8] A. Kagi, D. Burger, and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *International Symposium on Computer Architecture*, June 1997.

[9] P. Magnusson, A. Landin, and E. Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical Report T94:07, Swedish Institute of Computer Science, Kista Sweden, February 1994.

[10] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[12] Myricom. Myricom GM myrinet software and documentation. http://www.myri.com/scs/GM/doc/gm_toc.html, 2000.

[13] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99*, April 1999.

[14] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A non-uniform-memory-access programming model for high performance compuers. *The Journal of Supercomputing*, 10:197–220, 1996.

[15] J. Nieplocha, J. Ju, and E. Apra. One-sided communication on the Myrinet-based SMP clusters using the GM message-passing library. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS '01*, April 2001.

[16] J. Nieplocha, V. Tipparaju, A. Saify, and D. K. Panda. Protocols and strategies for optimizing remote memory operations on clusters. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC)*, April 2002.

[17] K. Parzyszek, J. Nieplocha, and R. Kendall. A generalized portable SHMEM library for high performance computing. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, 2000.

[18] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Science*, 7(1):61–77, 1989.

[19] V. S. Sunderam. PVM: A Framework for Parallel and Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[20] M. Trehel and M. Naimi. An improvement of the log(n) distributed algorithm for mutual exclusion. In *Proceedings of the IEEE International Conferece on Distributed Computer Systems*, pages 371–375, 1987.