NIC-Based Reduction in Myrinet Clusters: Is It Beneficial?[1]

Darius Buntinas Dhabaleswar K. Panda
Network-Based Computing Laboratory
Department of Computer and Information Science
The Ohio State University
{buntinas, panda}@cis.ohio-state.edu

***Abstract*— Reduction-to-one and reduction-to-all operations are common operations in parallel and distributed systems. These operations are *collective operations* which can involve many processes. It is therefore important to make these operations fast and efficient. Some modern network interface controllers (NICs) for system area networks (SANs) have programmable processors which can be used to offload protocol processing from the host processor. In this paper we investigate the use of the NIC processor to improve the performance of reduction operations. We implemented a NIC-based reduction-to-one operation which can perform integer and floating point operations, and evaluated our implementation. Our evaluation shows that the NIC-based operation performs better than the traditional host-based approach with up to a 1.19 factor of improvement. We also see that using NIC-based reduction can reduce host CPU utilization by a factor of improvement of 2.7, and can reduce the effects of process skew by a factor of improvement of up to 4.5.**

I. Introduction

Reduction-to-one and reduction-to-all operations are common operations in parallel and distributed systems. These operations are *collective operations* which can involve many processes. It is therefore important to make these operations fast and efficient. Research has been done to make these operations efficient by taking advantage of the particular characteristics of the underlying architecture [1], [2]. Some modern network interface controllers (NICs) for system area networks (SANs) have programmable processors which can be used to offload protocol processing from the host processor. Such implementations not only allow efficient communication operations, but also significant potential for overlap of computation with communication. Programmable NICs have been used to improve the performance of certain operations, as well as reduce the host involvement in the operations allowing the host to perform other useful computation [3], [4], [5], [6], [7]. However, none of these implementations have explored the benefits of NIC-based implementation for reduction for integer and floating point data. In this paper we explore the benefits of NIC-based support for reduction operations. It is worthwhile to note that a large fraction of reduction operations are performed using small data sizes of just a few elements. This means that a specialized reduction operation which can efficiently perform the operation on a small number of elements would be useful.

Using NIC-based collective communication operations, such as reduction, can significantly reduce host CPU utilization. This is because the NIC processor is performing the operation rather than the host processor. Another benefit of NIC-based operations is that they can make a parallel program using these operations less sensitive to process skew. Processes of a parallel program can become unsynchronized, or skewed, during the course of running the

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

application. This can happen as a result of unbalanced or asymmetric code, through random, unpredictable causes such as a process being context switched, or because the processes may not have been started at the same time. Such skew can have a significant impact on the performance of a parallel program when host-based collective communications operations, such as reduction are used. In a reduction operation, at a particular node, data from certain nodes must be received before the arithmetic operation can be performed and the result can be forwarded to other nodes. If this reduction operation is implemented at the application-level then upon calling the reduction function, the process must wait to receive all of the messages, performs the operation, and sends the result on. This means that if a process is delayed and hasn't performed the reduction operation, then another process may also be delayed waiting for that message, and so cannot continue with useful computation. However, because the NIC-based reduction operation is performed by the NIC, and not the host, once the process passes its data to the NIC, it can continue with its computation, and will not be stalled due to a delayed process.

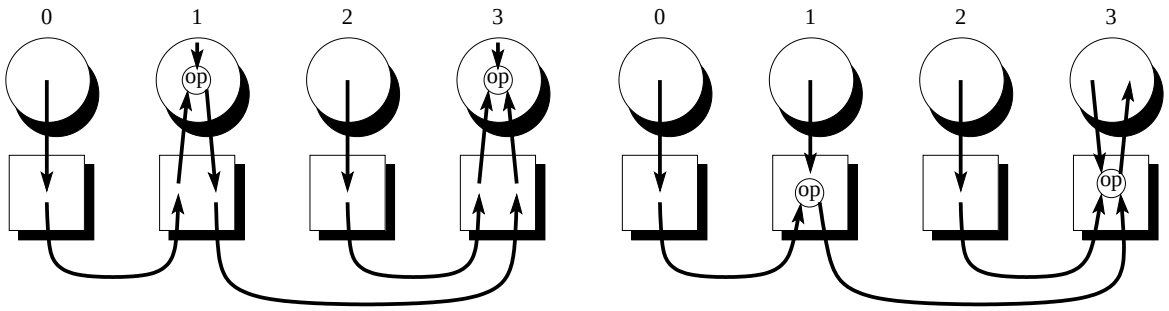
We have implemented a NIC-based reduction-to-one operation to perform integer and floating-point operations on single 64 bit elements. In this paper we describe the design and implementation of this operation as well as the evaluation of the implementation. Our implementation achieves a 1.19 factor of improvement over the traditional host-based implementation when performing integer operations on a 16 node system. We show that NIC-based reduction would also be beneficial for larger system sizes. Our evaluation also shows that using NIC-based reduction can reduce host CPU utilization by a factor of improvement of 2.7, and can reduce the effects of process skew by a factor of improvement of up to 4.5. Our initial implementation and evaluation indicates that reduction operations can benefit by NIC-based implementations, and that further work should be done to implement a more complete NIC-based reduction-to-one and reduction-to-all operations.

The rest of this paper is organized as follows. In the next section, we describe the general concept of a NIC-based reduction operation. In Section III we describe our design and implementation, followed by the evaluation of our implementation in Section IV. Finally we present our conclusions and future work in Section V.

II. NIC-Based Reduction

Before we describe the general concept of NIC-based reduction, we will briefly describe traditional host-based reduction. In traditional host-based reduction in a message-passing system, messages are passed between processes running at the host, and the arithmetic operations are performed by the host processors. When the operation is complete, the result of the operation will be located at one of the processes. Processes participating in the reduction operation are organized in a logical tree. Each process receives reduction messages from its children, which contain a partial result from the subtree of that child. Next, each process performs the arithmetic operation on its data and the partial results received from its children. The process then sends this result to its parent. Figure 1(a) shows a block diagram of a host-based reduction operation across four nodes. Node 0 sends its data to Node 1. When Node 1 receives this message it performs the arithmetic operation on the data from Node 0 and its data. Node 1 then sends this result to Node 3. Node 3 receives data from Node 2 and Node 1, and performs the arithmetic operation on its own data and the data sent by Nodes 1 and 2.

In a NIC-based reduction, each process sends its data to the NIC. The NIC will then wait for the messages from its children, perform the arithmetic operation, and either send a message to its parent, or if this node is the root of the tree and has no parent, it will forward the



(a) Host-based (b) NIC-based

Fig. 1. Block diagrams of Host-based and NIC-based reductions across four nodes. The circles represent the host processor of a node and squares represent the NIC of a node.

result to the host. Figure 1(b) shows a block diagram of a NIC-based reduction operation across four nodes. Here we see each process sending its data to the NIC. The NICs at Nodes 0 and 2 immediately forward their data to their parents, since they are leaf nodes. The NIC at Node 1 receives the message from Node 0 and performs the arithmetic operation on this data and the data sent from the host. It then sends this result to the NIC at Node 3. The NIC at Node 3 receives the messages from the NICs at Nodes 1 and 2, performs the arithmetic operation on this data and on the data sent from the host, then forwards this result to the host.

Notice that in the host-based reduction, messages received at intermediate nodes, such as Node 2, are received by the NIC, forwarded to the host, which performs the arithmetic operation and sends another message that is sent down to the NIC to be transmitted. In the NIC-based case, because the arithmetic operation is performed at the NIC, such messages do not have to be passed between the NIC and the host. Only the initial data needs to be passed from the host to the NIC. This can improve the performance of the reduction operation.

Another potential benefit of NIC-based reduction is reduced host involvement in the operation. For non-root nodes, once the host has sent its data to the NIC, it no longer has to be involved in the reduction operation. In the host-based case, the host process must either wait to receive the messages from its children, or it must be interrupted when the messages arrive so that it can perform the arithmetic operation on them. Since interrupts are time consuming operations, using these can lead to poor reduction latency, and is not commonly used for reduction operations.

However, the alternative of waiting for the messages has its own drawbacks. If processes are skewed, meaning that some processes are performing the reduction operation while others are lagging behind and have not yet started the operation, then intermediate processes may be waiting for other processes in their subtree to catch up. This can lead to poor overall application performance. By using NIC-based reduction the host needs only to supply the data to the NIC. It can then proceed on with other useful computation. This allows greater overlap of computation and communication operations. Furthermore, because the host is not involved in actually performing the operation, NIC-based reduction is a non-blocking operation. The root process need not wait idle for the result after it sends its data to the NIC. It can proceed with other computation and only get the result from the NIC when it needs it. This can further reduce host involvement.

III. Design and Implementation

We implemented our NIC-based reduction operation as a modification to the GM message passing system [8] which uses the Myrinet network [9], a popular system area network for clusters. Before we describe the design and implementation of our NIC-based reduction, we will give some background on GM and Myrinet.

Myrinet is a high-performance full-duplex 2Gbps network which uses NICs with programmable processors. GM is a user-level message passing system which uses the programmable NICs for much of the protocol processing. GM consists of three components: a kernel module, a user-level library, and a control program which runs on the NIC processor. When a user application wishes to send a message it calls the appropriate function from the library. This function constructs a *send descriptor* which describes what data is to be sent and to which process to sent it to. This descriptor is then written to the NIC using PIO. The NIC detects that a new descriptor has been written and processes it, DMAing the data from the host buffers and transmitting the message. In order to receive a message, the process must provide memory buffers in host memory into which the NIC will DMA the message data. This is done by sending the NIC a *receive descriptor* which describes such a buffer. When the NIC receives a message it DMAs the data into one of the buffers, then DMAs a notification to the host process that a message has been received. The host process can either poll for these notifications, or can block while waiting. In the latter case the NIC will signal an interrupt after it DMAs the notification.

We implemented a NIC-based reduction operation by modifying GM version 1.6.3. Our implementation can perform binary *AND* and *OR* operations, as well as integer and floating point *SUM*, *MIN* and *MAX* operations on a single 64 bit data element. The host process passes a descriptor to the NIC describing the reduction operation. As the NIC receives reduction messages from the network, it performs the arithmetic operation on the data and stores the result. Once messages from all of the children have been received and processed, if the process initiating the reduction operation is the root, the NIC DMAs a notification to the host, including the result, indicating that the reduction has completed, otherwise, the NIC transmits the result to the parent NIC.

There are several design issues in this implementation, namely, dealing with unexpected messages, dealing with multiple instances of the reduction operation, generating and specifying the tree structure, and performing floating point operations at the NIC. In the rest of this section we will discuss these issues.

A. Unexpected messages

Because processes are not always synchronized, it is possible that some processes may execute the reduction operation before others. This means that a NIC may receive reduction messages from other NICs before the host process has initiated the reduction operation and send its data. Since the host has not informed the NIC which processes to expect data from, and what arithmetic operation to perform, the NIC cannot process the messages. Such a message can be handled in one of two ways. One option is to reject the message and request that the sender retransmit it later. Another option is for the NIC to store the data until the host has initiated the corresponding reduction operation. The first option can lead to high latency because the messages need to be retransmitted after a delay. While the second option gives better performance, it requires NIC memory to be allocated for storing this data. Since NIC memory is limited, this may limit the number of messages that can be stored.

We used a hybrid approach where we provided a limited number of buffers to store unexpected data, and reject messages once these are full. When the NIC receives a descriptor from the host for a reduction operation, it

checks the list of unexpected messages. If it finds any unexpected messages that match, it performs the operation on that data, and frees that unexpected message buffer.

B. Multiple instances of the reduction operation

When a non-root process initiates a reduction operation, after it sends the data to the NIC, it can proceed with its computation. This means that a process can initiate a second reduction operation before the NIC has completed the first. The NIC needs to be able to process multiple instances of the operations in the correct order. We did this by keeping a queue of instances of reduction operations for each host process. When a reduction message is received from the network for a particular process, the NIC searches the list of instances for that process looking for a matching instance. If a matching instance is found, the arithmetic operation is performed for that instance, otherwise the message is considered an unexpected message and is handled as described above.

C. Generating and specifying the tree structure

The tree structure can be generated by either the NIC or the host process. However, because NIC processors are typically much slower than host processors, it would be more efficient to have the host construct the tree and pass a list of children and the parent to the NIC. We used this option. The send descriptor was only 64 bytes so we are limited as to the number of children that can be specified. Four bytes are needed for each child: two bytes are needed to specify a GM node, one byte is needed to specify the GM *port*, and one byte is used in the algorithm to indicate whether a reduction message has been received from this process. Since we also include the eight-byte data in the descriptor, and 12 more bytes are used in the descriptor for other fields, there is only room to specify nine children, and one parent.

The shape of the reduction tree is also an important design issue. The latency of the operation increases with each level of the tree, so a very deep tree may not be desirable. On the other hand, a very shallow tree increases network contention as many child nodes transmit their data to one parent node. The exact shape of the tree depends on the performance characteristics of the reduction operation. We have not fully investigated the optimal tree shape for NIC-based reduction. For our evaluation we used a binomial tree because this is the most common tree used for reduction operations, e.g., MPICH [10] uses a binomial tree.

D. Performing floating point operations at the NIC

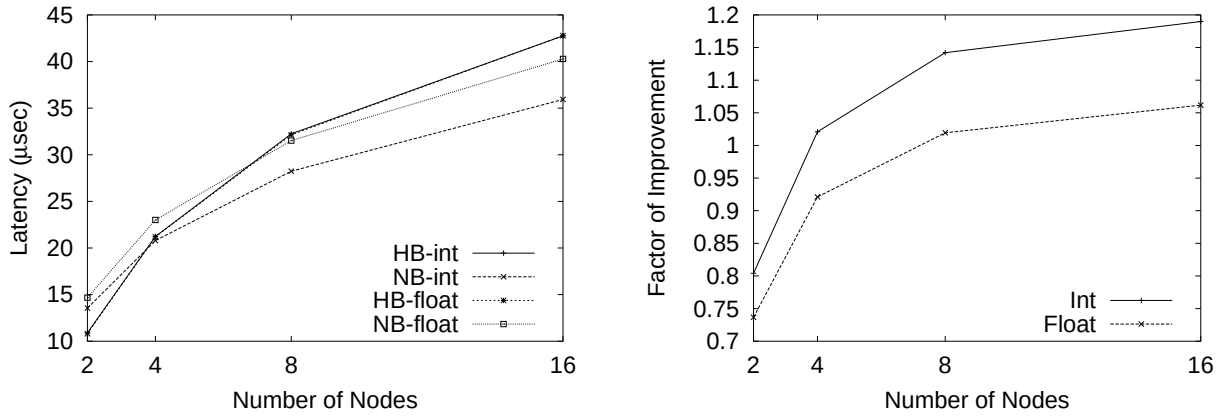
The Myrinet NIC processors do not have floating point units. So in order to be able to perform floating point operations, we had to use floating point operations implemented in software. We used the SoftFloat [11] library for these operations. SoftFloat is a free software implementation of the IEC/IEEE Standard for Binary Floating-point Arithmetic, and supports all functions dictated by the standard for 32, 64, and 128 bit floating point formats. We used only the 64 bit format in our implementation.

IV. Experimental Results

In this section, we evaluate our implementation on a cluster of 16 quad-SMP 700MHz Pentium-III nodes with 66MHz/64bit PCI. The nodes are connected to a Myrinet2000 network. The NICs are PCI64B cards with 2MB of memory and 133MHz LANai 9.1 processors. These are connected to 16 ports of a 32 port switch. We compare our NIC-based reduction implementation, which is based GM version 1.6.3, to a host-based reduction implementation using the same version of GM. We evaluate the basic reduction operation, the host CPU utilization of the reduction operation, and its tolerance of process skew.

A. Basic reduction

To evaluate the performance of our reduction implementation, we compare the time from



(a) Latency

(b) Factor of Improvement

Fig. 2. Comparison of NIC-based reduction (NB) and host-based reduction (HB) for integer (int) and floating-point (float) operations

when the leaf node furthest away from the root initiates the operation until the root node receives the result. We performed the test in the following manner. All of the nodes perform the reduction operation. As soon as the root node completes the operation and receives the result, it sends a message to the last leaf node of the tree. Once this node receives the message it takes the time between when it initiated the reduction operation and when it received the message, then subtracts off the one way latency time. We take the average time over 10,000 iterations.

We performed the evaluation for 2, 4, 8 and 16 nodes using integer operations and floating-point operations. Figure 2 shows the results of this evaluation. The figures show the results for the integer and floating point SUM operation. Notice also that the results for the host based floating point and integer operations were very similar, so the two lines on the graph are on top of one another. The graphs show that for integer operations, NIC-based reduction performs better than host-based for all but the two node case. We see that the floating-point operations add some overhead, but that NIC-based reduction is still better than the host based reduction for all but the two and four node cases. We see up to a 1.19 factor of improvement for the integer operation, and up to a 1.06 factor of improvement for floating point operations.

B. Larger system sizes

The factor of improvement for NIC-based reduction increases with the number of nodes. This indicates that for larger system sizes, the NIC-based reduction operation may be even more beneficial. In order to investigate how the relative performance of NIC-based reduction would change with an increase in system sizes we compared the performance of the operations using a 1-degree tree, in other words a chain, and varied the depth of the tree. Figure 3 show the results of this comparison. Notice again in this graph that the lines for host-based floating-point and integer operations overlap. This graphs shows us that as the depth of the tree increases the latency of the host-based operation increases faster than the NIC-based operation. In fact the time for host-based integer reduction increases at a rate of 3.70µs per level of depth faster than that for NIC-based integer reduction. Similarly, the latency of host-based floating-point reduction increases

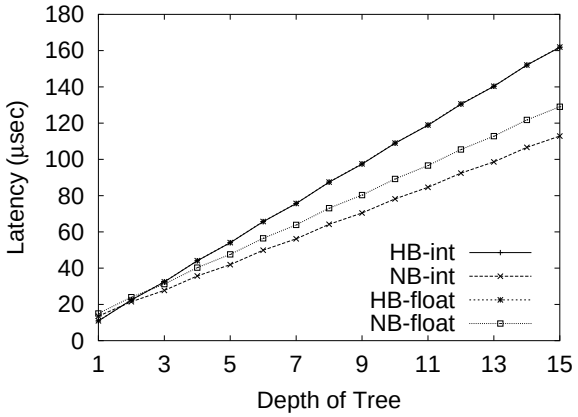


Fig. 3. Latency of NIC-based reduction (NB) and host-based reduction (HB) for integer (int) and floating-point (float) operations using a 1-degree tree (a chain) of varying depth

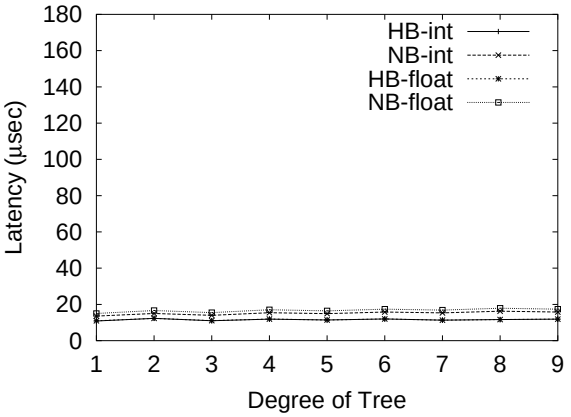


Fig. 4. Latency of NIC-based reduction (NB) and host-based reduction (HB) for integer (int) and floating-point (float) operations using trees of depth 1 with varying degree

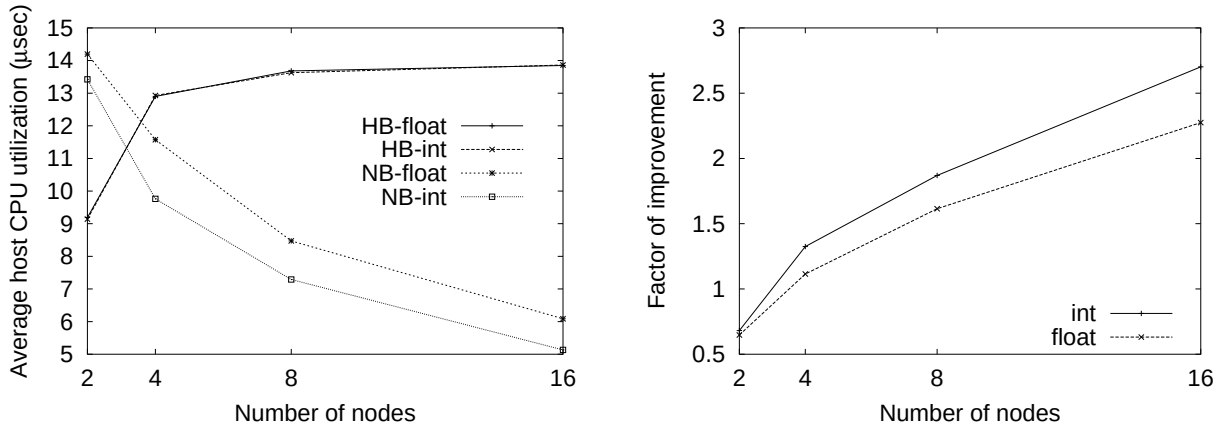
at a rate of 2.64µs per level of depth faster than that of NIC-based floating-point reduction. We see that for a tree of depth 1 host-based reductions perform better than NIC-based reductions. Similarly, host-based floating-point reduction performs better than NIC-based floating-point reduction for the tree of depth 2. We believe that this is because of the overhead of the more complicated operation at the slower NIC processor. As the depth increases, the number of times messages have to be sent between the NIC and the host at intermediate nodes increases in host-based reduction. Since NIC-based reduction avoids this overhead, it performs better for deeper trees.

As system sizes increase, and trees get larger, the maximum degree of the tree also increases. To study the effect of increasing the degree of a tree, we compared the latency of NIC-based and host-based reduction operations for trees of depth 1 with varying degree. Figure 4 shows the results of this test. Again the host-based floating-point and integer lines overlap. We see here that host-based reductions perform better than NIC-based for any of the trees. However host-based integer reduction performs only about 2.19µs better, and host-based floating-point reduction performs only 3.63µs better. Furthermore, while there is a slight increase in overhead for NIC-based reductions as the degree of the tree increases, it is quite small, 0.22µs per degree for the integer reduction, and 0.24µs per degree for the floating-point reduction. For trees such as binomial trees the depth of the tree increases at the same rate as the depth of the tree. This indicates that the NIC-based reduction will continue to perform better than the host-based reduction for large system sizes.

C. Host CPU utilization

One of the major benefits of NIC-based reduction is that it reduces the load on the host processor. Once the host process sends its data to the NIC, it is free to perform useful computation. To compare the host CPU utilization for NIC-based and host-based reduction, we timed how long the host process spends performing the reduction at each node. Our test consists of each process performing a barrier synchronization followed by a reduction operation. Figure 5 shows the average of 10,000 iterations of this test.

In Figure 5(a) we see the average host CPU utilization for NIC-based and host-based reduction for various numbers of nodes. Notice that for all but two node reductions, NIC-based reductions use the host CPU less than host-based reductions. Also note that the average CPU



(a) Average host CPU utilization

(b) Factor of Improvement

Fig. 5. Average time spent by the host processor performing the reduction for host-based (HB) and NIC-based (NB) reductions performing integer (int) and floating-point (float) operations

utilization for host-based reduction increases as the number of nodes increases. This is because, in the reduction tree, as the number of nodes increases, there are more interior nodes, which are waiting for reduction messages from their children. In NIC-based reduction, we see that the average host CPU utilization actually decreases as the system size increases. This is because for non-root nodes, the host process simply has to construct the send descriptor and send it to the NIC. So regardless of how many nodes are performing the reduction, the host CPU utilization at non-root nodes is only a few hundred nanoseconds. In this test, the root node waits for the result of the reduction after it sends its data to the NIC, so the CPU utilization for the root node will increase with the latency to perform the reduction, as it does for host-based reduction. However, as the number of nodes increases, the CPU utilization at the root increases slower than the number of nodes performing the reduction, so the *average* host CPU utilization decreases with the number of nodes.

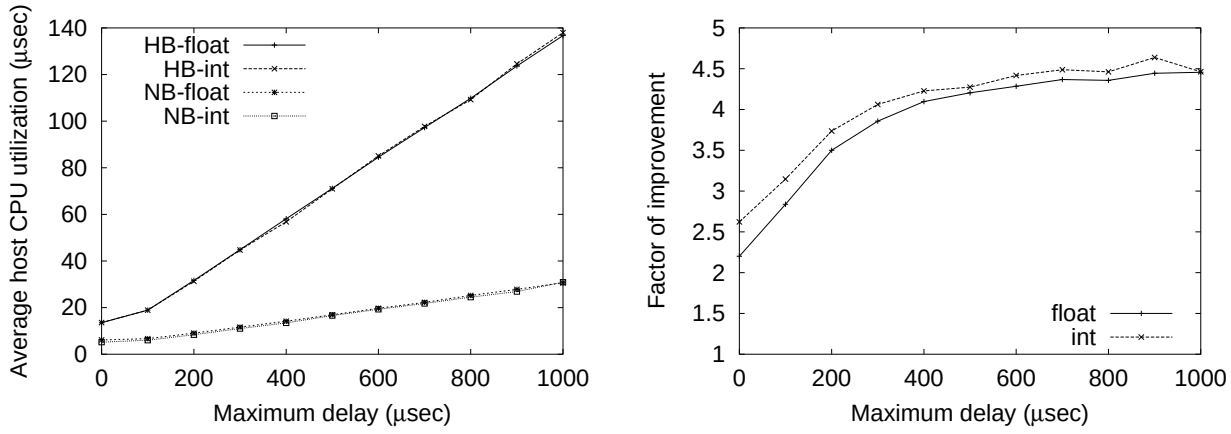
Note, that NIC-based reduction allows for a *non-blocking* implementation, where the root process does not wait for the result from the NIC after it sends its data. Instead, the process can go on to perform other useful computation, that does not depend on the result, and it would only read the result from the NIC once it needs that data. This would allow a further reduction in CPU utilization.

Figure 5(b) shows the factor of improvement in CPU utilization for NIC-based reduction over host-based reduction. We see a factor of improvement of 2.7 for integer operations and 2.3 for floating point operations. Notice that the factor of improvement increases as the system size increases.

D. Tolerating process skew

Another major benefit of NIC-based reduction over host-based reduction is its tolerance to process skew. In order to see what effect process skew has on host CPU utilization, we timed how long each host process spends performing the reduction operation, while varying the skew between processes.

In this test, the processes perform a barrier synchronization, followed by a delay, the length of which is chosen at random between 0 and a maximum delay value. The processes then perform a reduction operation. This is repeated 10,000 times. By varying the maximum delay



(a) Average host CPU utilization

(b) Factor of Improvement

Fig. 6. Average time spent by the host process performing the reduction, with different levels of process skew for host-based (HB) and NIC-based (NB) reductions performing integer (int) and floating-point (float) operations

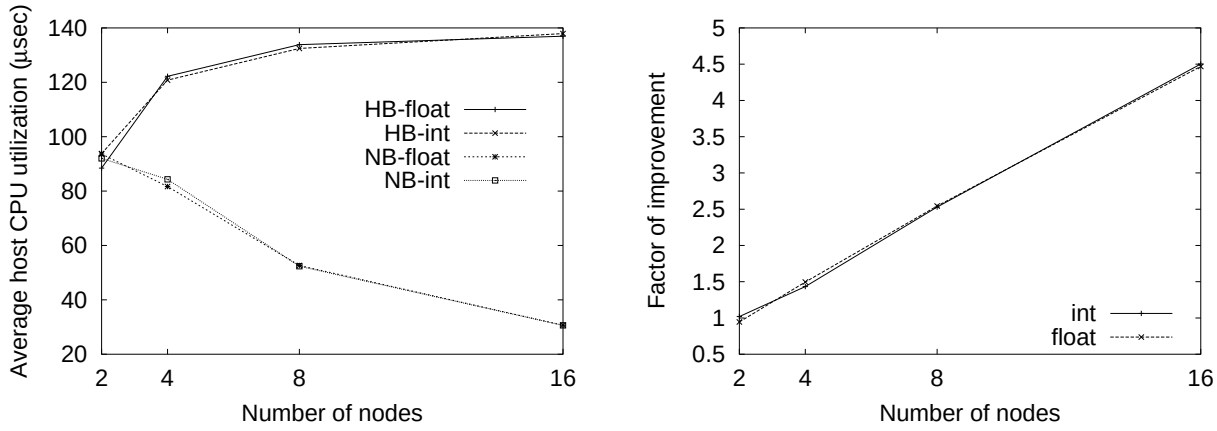
value, the level of skew between processes varies. As the maximum delay value increases, the skew between processes increases.

The results of this test are shown in Figure 6. Figure 6(a) shows the average time spent by all of the host processes of 16 nodes performing the reduction, as the maximum delay value is varied. Recall, that in a tree-based reduction, a node must receive all messages from its children, as well as provide its own data, before it can pass the result on to its parent. When a child node or, more generally, any descendant node is delayed, the reduction operation at that node cannot proceed. We would expect, then, that as process skew increases, the number of descendants of a process which are delayed increases, as well as the amount by which they are delayed. For host-based reduction, where the host processes must wait for messages from child processes, process skew results in processes that are stalled and cannot perform useful computation.

We see, in Figure 6(a), that for host-based reduction, as the maximum delay increases, the CPU utilization increases dramatically. However, for NIC-based reductions, only the root node is delayed by skewed processes. Because the reduction operation is performed at the NIC, non-root processes simply have to pass their data to the NIC. This makes non-root processes unaffected by process skew. In Figure 6(a), we see process skew has relatively little effect on the average time host processes spend on reduction.

Figure 6(b) shows the factor of improvement in CPU utilization for NIC-based reduction over host-based reduction. We see up to a 4.5 factor of improvement for both integer operations and floating point operations. Furthermore we see significant improvements even when process skew is small. For instance we see a 3.7 factor of improvement for integer operations even when the maximum delay value is only 200µs .

We also evaluated the effect of system size on host CPU utilization by fixing the skew and varying the number of nodes performing the reduction. In Figure 7(a) we see that as the number of nodes increases, the average host CPU utilization for host-based reduction increases. However, for NIC-based reductions, we see that the average host CPU utilization decreases. This is because in the NIC-based case, only the root process is affected by process



(a) Average host time

(b) Factor of Improvement

Fig. 7. Average time spent by the host process performing the reduction, with a maximum delay value of 1000µs , for different system sizes for host-based (HB) and NIC-based (NB) reductions performing integer (int) and floating-point (float) operations

skew. As the number of processes increases the time spent by the root process contributes a smaller fraction to the average. For the host-based case, each process can be affected by skew, and as the number of processes increases, there are more processes which can be delayed, relative to their parents and ancestors, so the average host CPU utilization increases as the system size increases. Figure 7(b) shows the factor of improvement for NIC-based reduction over host-based reduction. We see up to a 4.5 factor of improvement for both integer operations and floating point operations.

These results indicate that NIC-based reduction operations are much more tolerant to process skew than host-based reduction. Furthermore we see that process skew has a greater impact on host-based reduction as system size increases. This means that using NIC-based reduction can significantly increase the scalability of a system in the presence of process skew.

V. Conclusions and Future Work

We have presented an initial implementation of a NIC-based reduction operation, and evaluated it. We found up to a 1.19 factor of improvement for integer reduction and 1.06 factor of improvement for floating-point reduction. We also give evidence that NIC-based reduction will perform better than host-based reduction in larger systems. Though this improvement is not very large, the fact that the operation does not involve the host allows useful computation at the host to be overlapped with the reduction operation at the NIC. In fact, we have shown a 2.7 factor of improvement in CPU utilization when using NIC-based reduction for integer operations and a 2.3 factor of improvement when using floating point operations. We further note that NIC-based reduction can be used in a non-blocking fashion which would further improve the CPU utilization.

We have also shown that NIC-based reduction is much more tolerant to process skew than the host-based implementation. In the presence of process skew NIC-based reduction gives a 4.5 factor of improvement in CPU utilization over host-based reduction. We also noticed that when the system size increases, the effect of the skew impacts host-based reduction much more than the NIC-based reduction. This indicates

that NIC-based reduction would improve the scalability of certain applications.

Overall these results indicate that NIC-based reduction can perform better than host-based reduction, can improve CPU utilization, and can tolerate process skew better than host-based reduction. This leads us to conclude that further work should be done in this direction. We intend to improve NIC-based reduction operation to allow for multiple elements, up to 64 elements of 64 bits. We also intend to increase the maximum number of children from the current limit of nine children. Another issue that needs to be addressed is the order in which the arithmetic operations are performed. Currently the arithmetic operations are performed on the data in the order in which the messages arrive at the NIC. This may change from one run to the next. Because of the potential of rounding errors, overflow and underflow in floating-point operations, this could lead to the reduction operation giving different results for the same input. By fixing the order in which the operations are applied to the data, the result will be deterministic. We also plan to modify MPICH-GM to use NIC-based reduction, and evaluate a non-blocking version on `MPI_Reduce()`. Finally, we intend to implement a NIC-based reduction-to-all operation.

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) and Parallel Architecture and Communication Group (http://www.cis.ohio-state.edu/∼panda/pac.html).

References

[1] R. A. V. de Geijn, "On Global Combine Operations," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 324–328, 1994.

[2] D. K. Panda, "Global Reduction in Wormhole k-ary n-cube Networks with Multidestination Exchange Worms," in *International Parallel Processing Symposium*, Apr 1995, pp. 652–659.

[3] K. Verstoep, K. Langendoen, and H. Bal, "Efficient Reliable Multicast on Myrinet," in *Proceedings of the International Conference on Parallel Processing*, Aug 1996, pp. III:156–165.

[4] 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)*, August 1998, pp. 381–390.

[5] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan, "Broadcast/Multicast over Myrinet using NIC-Assisted Multidestination Messages," in *Proceedings of the International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC)*, 2000, pp. 115–129.

[6] 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.

[7] 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.

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

[9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. Seizovic, and W. Su, "Myrinet - a gigabit per second local area network," in *IEEE Micro*, February 1995.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sept. 1996.

[11] J. Hauser, "SoftFloat," http://www.cs.berkeley.edu/∼jhauser/arithmetic/SoftFloat.html.