GPU-Aware Intranode MPI_Allreduce

Iman Faraji
ECE Dept., Queen's University
Kingston, ON, Canada K7L 3N6
i.faraji@queensu.ca

Ahmad Afsahi
ECE Dept., Queen's University
Kingston, ON, Canada K7L 3N6
ahmad.afsahi@queensu.ca

ABSTRACT

Modern multi-core clusters are increasingly using GPUs to achieve higher performance and power efficiency. In such clusters, efficient communication among processes with data residing in GPU memory is of paramount importance to the performance of MPI applications. This paper investigates the efficient design of intranode MPI_Allreduce operation in GPU clusters. We propose two design alternatives that exploit in-GPU reduction and fast intranode communication capabilities of modern GPUs. Our GPU shared-buffer aware design and GPU-aware Binomial reduce-broadcast algorithmic approach provide significant speedup over MVAPICH2 by up to 22 and 16 times, respectively.

Keywords

MPI, GPU, Intranode MPI_Allreduce, Shared Buffer, IPC

1. INTRODUCTION

While multi-core processors have become an integral part of clusters, equipping cluster nodes with GPU accelerators have shown to be a promising approach to achieve higher performance, improved performance-per-watt and better compute density. Many of today's High-Performance Computing (HPC) clusters employ such hybrid architectures [16].

In traditional clusters, the Message Passing Interface (MPI) [7] is the programming paradigm of choice for high-end computing. In GPU clusters, while computation can be accelerated by being offloaded to the GPUs, processes with data residing in GPU memory still require support from MPI library for communication. It has been shown that intranode and internode communications between GPUs in HPC platforms play an important role in the performance of scientific applications [1, 10]. In this regard, researchers have started looking into incorporating GPU-awareness into the MPI library, targeting both point-to-point and collective communications [12, 5, 14, 11].

In this paper, our focus is on the efficient design and implementation of GPU-aware intranode MPI_Allreduce collective operation. MPI_Allreduce has been shown to have high share of execution time among collective operations in MPI applications [13]. Any performance improvement in this operation may significantly enhance the performance of such applications. MPI_Allreduce is a computational collective operation; reductions are data parallel operations that can highly benefit from the massive number of threads in modern GPUs. We are targeting the intranode phase of a hierarchical MPI_Allreduce. Various algorithms [15] have been proposed for internode MPI_Allreduce in traditional clusters that can be directly applied to the internode stage of a hierarchical approach in GPU clusters.

Current intranode GPU-aware MPI_Allreduce approaches require staging the GPU data in host memory for a follow-up CPU-based MPI_Allreduce operation, and then moving the result back to the GPU memory. This approach is inefficient, as on one hand, it requires moving the data back and forth between the GPU and the host memories. On the other hand, it does not use the GPU computational capability in performing the reduction operations.

NVIDIA, in CUDA 4.1 [9], introduced the Inter-Process Communication (IPC) feature, which allows direct intranode GPU-to-GPU communication. This way, there is no need to stage the GPU data in and out of the host memory, which can significantly enhance the performance of intranode inter-process GPU-to-GPU communication. Previous research has used CUDA IPC to optimize point-to-point and one-sided communications in MPI [12, 5]. However, to the best of our knowledge, CUDA IPC has not been used in the design of collective operations.

This paper contributes by proposing two design alternatives for a GPU-aware intranode MPI_Allreduce that, for the first time, perform the reduction operations within the GPU and leverage CUDA IPC for communications among processes involved in the collective operation. The first approach, called the *GPU shared-buffer aware* MPI_Allreduce, gathers the pertinent data into a shared buffer area inside the GPU global memory, does an in-GPU reduction on the gathered data, and then copies the reduced data to other processes. The second approach, called the *GPU-aware Binomial reduce-broadcast* MPI_Allreduce, uses a Binomial reduction tree to perform in-GPU reduction operations, followed by a broadcast operation.

We have implemented the proposed techniques in MVAPICH2 [8]. Our experimental results show that our GPU shared-buffer aware MPI_Allreduce is superior over the GPU-aware Binomial reduce-broadcast design in all test cases. The proposed designs show up to 22 and 16 times speedup

EuroMPI/ASIA '14, September 9-12 2014, Kyoto, Japan

http://dx.doi.org/10.1145/2642769.2642773.

over existing MPI_Allreduce in MVAPICH2, respectively.

The remainder of this paper is organized as follows. In Section 2, we provide the background material for this work. In Section 3, we discuss the design and implementation of our GPU-aware MPI_Allreduce approaches. We present our experimental results in Section 4, and discuss the related work in Section 5. Finally, in Section 6, we make concluding remarks and outline our future work.

2. BACKGROUND

2.1 GPU Architecture and Programming

A modern GPU architecture is assembled of an array of Streaming Multiprocessors (SM); SMs are composed of a set of Streaming Processors (SPs). There are thousands of SPs in a single GPU. GPUs are highly multi-threaded and parallel execution on GPUs is handled by these fine-grained threads. The performance of a modern GPU is at least 1 teraflop per second on single/double precision floating-points. As a result, GPUs can highly benefit data-parallel and compute-intensive kernels of the applications.

Compute Unified Device Architecture (CUDA) [3] is the software framework for programming NVIDIA GPUs. CUDA is an extension over the ISO C, developed by the NVIDIA Corporation. A CUDA code running on the GPU is called a kernel, which is compiled by the NVIDIA CUDA Compiler (NVCC). The kernels execute in a multi-threaded fashion with the SIMD model.

2.2 MPI_Allreduce

The Message Passing Interface (MPI)[7] is the most prevailing programming model in HPC clusters. MPI basically has three communication models: point-to-point, collectives, and Remote Memory Access (RMA). MPI collectives, which involve communications among a group of processes, play a crucial role in the performance of MPI applications. They can be implemented in a hierarchical or non-hierarchical fashion.

MPI_Allreduce is a collective operation that performs the reduction operation on a distributed dataset among processes, and stores the result back in all of the participating processes. Various algorithms have been proposed for this routine. The performance of these algorithms depends on various factors, such as message size and number of processes. The Binomial reduce followed by a broadcast, recursive doubling and Rabenseifner algorithms are examples of such algorithms [15].

The existing implementation of MPI_Allreduce in MVAPICH2 moves the pertinent data from the GPU memory to the host memory; then, a CPU-based MPI_Allreduce is performed on the data residing on the main memory; and finally, the result moves back to the GPU memory. This approach requires costly GPU-to-CPU and CPU-to-GPU copies, and does not leverage any kernel function to perform the reduction operations within the GPU.

2.3 CUDA Inter-process Communication

Any device memory or event created by a host thread can be directly accessed by the same host thread or threads within the same process. Threads belonging to other processes cannot directly reference these memory areas or events. In order to access the GPU data in remote processes, NVIDIA in CUDA 4.1 [9], has introduced CUDA Inter-process Communication API, which allows device memory pointers and events to be shared among processes on the same node.

To use CUDA IPC, first an IPC handle for a given device memory pointer or event has to be created using `cudaIpcGetMemHandle()` or `cudaIpcGetEventHandle()`, respectively. The handle can be passed later on to the importing processes using standard communication features, with host staging. Such processes will then be able to retrieve the memory pointer or event using `cudaIpcOpenMemHandle()` or `cudaIpcOpenEventHandle()`, respectively.

When an importing process opens the memory handle, it returns a device pointer that would allow the address space of the exporting process to be mapped to its own address space. Although the importing process can access and modify the exporting address space, the completion of an IPC copy operation cannot be guaranteed until determined by IPC events. For that, an IPC event has to be shared between the importing and exporting processes. The IPC events must be created with the `cudaEventInterprocess` and `cudaEventDisableTiming` flags set, in order to be shared by processes.

3. DESIGN AND IMPLEMENTATION

Currently, GPU-aware MPI_Allreduce implementations in MPI libraries, such as MVAPICH2 [8], are capable of detecting that a pointer being passed to this operation is a device memory pointer. However, as stated earlier, the current GPU-aware approach used to implement MPI_Allreduce is inefficient as it has the overhead of copying the result back and forth between the device and the host memories; and it does not benefit either from the massive fine-grained parallelism in GPUs that can be specifically a good fit for reduction operations.

In our proposed designs, we perform direct GPU-to-GPU inter-process communication, eliminating staging the data in the host buffers by using GPU shared buffers and leveraging CUDA IPC feature. We also perform in-GPU element-wise reduction once the pertinent data becomes available in the GPU shared buffers.

3.1 GPU Shared-Buffer Aware MPI_Allreduce

In this design, all of the intranode processes copy their pertinent data residing on the GPU to a common shared buffer area inside the GPU global memory so that the reduction operation could take place by the GPU afterwards, as shown in Fig. 1. This shared buffer is a pre-allocated area in the address space of a predefined process (without loss of generality, we assume process with `rank 0` as the predefined process), which is then exposed to other processes. Processes exploit CUDA IPC to communicate through the GPU shared buffer region. Processes on the node also have access to a shared directory, which is responsible for indicating the completion of IPC copy operations. The shared directory can be allocated either on the GPU global memory or on the host memory. We have evaluated both design alternatives, and decided to keep the directory on the host memory (please refer to Section 3.3 for more details).

The size of the shared directory is `intra_comm_size` bits, in which `intra_comm_size` represents the number of processes on the node. Each of the first `intra_comm_size - 1` bits (`copy flags` - see Fig. 1) is associated with one process rank, and is set once that process initiates its IPC copy into the GPU shared buffer. The last bit in the directory

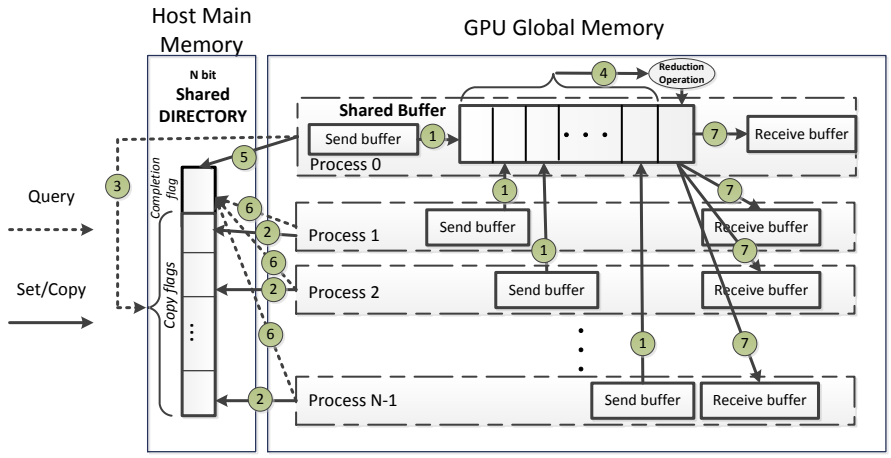

Figure 1: Steps of the GPU shared-buffer aware approach for MPI_Allreduce.

(`completion flag`) indicates the completion of the reduction operation and the availability of the results in the GPU shared buffer. This bit is set by process with `rank 0`.

The GPU shared buffer area should be sufficiently large to hold the gathered dataset from all intranode processes (256 MB in our experiments). This is directly related to the number of processes/GPUs per node, as well as the size of the dataset that are typically in use in applications leveraging MPI_Allreduce. As such, the size of allocated GPU shared buffer is much less than the amount of global memory available in modern GPUs. Therefore, this is not a scalability concern in our design.

Basic Steps of the Design: Fig. 1 depicts the steps involved in the GPU shared-buffer aware MPI_Allreduce design. These steps are as follows:

`Step 1.` All processes copy their share of data from their send buffers into their associated addresses in the GPU shared buffer area.

`Step 2.` All processes (except `process 0`) set their associated `copy flags` in the shared directory after initiating their IPC copies.

`Step 3.` `Process 0` waits on all `copy flags` to be set (this step can overlap with `Step 2`).

`Step 4.` Once all `copy flags` are set, all pertinent data is available in the GPU shared buffer. `Process 0` then performs in-GPU element-wise reduction on the aggregated data, and will then place it into the predefined location inside the GPU shared buffer.

`Step 5.` Once in-GPU reduction completes, `process 0` toggles the `completion flag` to inform other processes that the reduction operation is completed.

`Step 6.` All processes (except `process 0`) query the `completion flag` (this step can overlap with `Steps 3, 4,` and `5`).

`Step 7.` Once the `completion flag` is toggled, all processes copy the result into their respective receive buffers.

Note that in `Step 5`, the `completion flag` has to be toggled in each instance of MPI_Allreduce call; otherwise, successive MPI_Allreduce calls may end up reading stale data.

Implementation Details: The GPU shared-buffer aware approach leverages pre-allocated CPU and GPU shared buffers. The CPU shared memory region is allocated during MPI initialization stage, `MPI_Init()`, and is attached to the address space of intranode processes. The memory handle of the GPU shared buffer is also created, broadcast and mapped to the address space of other processes during this stage. This is done only once to mitigate the costly operation of exposing the GPU shared buffer address space.

IPC copies in CUDA have an asynchronous behavior, even if synchronous CUDA memory copy operations are issued on the shared region. Therefore, the `copy flags` in the shared directory only indicate the initiation of the IPC copies, but not their completion. To guarantee the completion of the IPC copies, we leverage the IPC events. The IPC event can be shared and used among processes residing on the same node. To share an IPC event, the handle of the allocated event is created and passed on by the predefined process to the other processes. To guarantee the completion of the IPC copies, immediately after each IPC copy (`Step 1` in Fig. 1), an IPC event is recorded using `cudaEventRecord()`, and then the associated `copy flag` in the shared directory is set (`Step 2`). Once all `copy flags` are set, it is an indication that all of the IPC events are recorded. Once `process 0` finds out that all `copy flags` have been set, it issues `cudaStreamWaitEvent()` on the IPC event which waits on the final recorded event. The process blocks on the `cudaStreamWaitEvent()` until the last recorded IPC event on the other processes reports completion. This way, the completion of the IPC copy operations will be guaranteed. Finally, all of the allocated shared buffers and events are freed and destroyed in the `MPI_Finalize()` call.

3.2 GPU-Aware Binomial Reduce-Broadcast MPI_Allreduce

In order to compare a well-known MPI_Allreduce algorithm against our GPU shared-buffer aware design, we have opted to implement the Binomial tree algorithm (for the reduction step) followed by a broadcast operation (Binomial reduce-broadcast). In the following, we try to articulate this decision. The Binomial reduce-broadcast and the Rabenseifner [15] are two of the well-known MPI_Allreduce algorithms. In the Binomial tree algorithm, the full vector is passed and the distance between communicating processes doubles after each step; it takes $\lceil log(n) \rceil$ steps (n is the

number of intranode processes) to reduce the data. In the Rabenseifner algorithm, MPI_Allreduce is performed by a reduce-scatter followed by an allgather operation. In the reduce-scatter phase, each process performs a reduction operation on the received data in each step of the algorithm. On the other hand, the Binomial tree reduction has much less number of reductions, though the vector size is larger. We note that the timing difference between GPU reduction operations on large and short vectors was not a bottleneck in our approach. We also had the same observation on in-GPU copies; in fact, it is better to have less number of copies with larger messages on the GPU instead of having more number of copies with smaller messages. This effectively means that there are fewer serializations on the GPU in the case of the Binomial algorithm. All in all, considering that computation and communication by different processes cannot be parallelized on the GPU due to the current CUDA IPC limit, we believe that the Binomial reduce-broadcast might be a more suitable algorithmic approach for the GPUs.

In the Binomial reduce-broadcast approach, we again use both GPU and host shared buffers. IPC events are similarly used in conjunction with a directory to indicate the completion of the IPC copies. In this approach, unlike the GPU Shared-buffer aware approach, the GPU shared buffer is distributed among processes for inter-process communication.

Basic Steps of the Design: For the sake of brevity, we only present the basic steps involved in the first tree level of our GPU-aware Binomial reduce for MPI_Allreduce.

`Step 1.` All processes copy the data in their send buffers into their receive buffers (receive buffers are used as temporary space to hold the intermediate reduction data).

`Step 2.` Odd processes IPC copy the contents of their receive buffers into the GPU shared buffers of their adjacent even processes.

`Step 3.` Odd processes set their associated `copy flags` in the directory after initiating their IPC copies.

`Step 4.` Even processes query the directory (this step can overlap with `Step 3`).

`Step 5.` Once `copy flags` are set, each even process performs an element-wise reduction between the data in its GPU receive buffer and the pertinent data in its shared buffer. The reduced result is stored back into the receive buffer and the algorithm proceeds to the next level.

This process continues for the next tree levels following the Binomial tree algorithm, until `process 0` performs the last reduction operation; it then stores the result into its GPU shared buffer and toggles the `completion flag`. In the meantime, other processes are querying the `completion flag`, and once set they copy the result into their own receive buffers. It is worth mentioning that it is possible to delay the reduction operation to a later tree level (e.g., one reduction in every two or three levels); however, this requires larger shared buffers to hold the intermediate data. Our evaluation has shown that this has a negligible impact on performance.

Implementation Details: As discussed earlier, in this approach we use shared buffers both on the host and on the GPU. The shared buffer on the GPU is allocated by only half of the participating processes. Thus, the size of the shared buffer in this approach is half the size of the shared buffer in the GPU shared-buffer aware approach. The directory is allocated on the host memory and its size is also half the size of the directory in the GPU shared-buffer aware approach. The memory and event handles are created and passed along during the MPI initialization time.

In this algorithm the order of communications and computations are guaranteed by enforcing processes belonging to the same tree level to only communicate with each other. The tree level of each sending process is stored in its associated entry in the shared directory. This way, the sending and receiving processes can check and match their tree levels before triggering IPC copies. Initiation of IPC copies in each tree level indicates that all of the copies in previous tree levels are completed. Hence, ordered communication/computation are guaranteed, which prevents the potential of any race condition.

3.3 Other Design Considerations

Both of the designs proposed in this paper allocate the shared buffer on the GPU while the directory is kept on the host main memory. In the first glance having the directory on the GPU memory with a kernel function querying its entries seems to be justified; however, this can potentially lead to spin-waiting on the directory forever, as the process querying the directory will take over the GPU resources and would prohibit other processes to access them. We tried to address this issue by forcing the querying process to release the GPU in time-steps. However, the performance results were not promising, and selecting the appropriate value for the time-step was dependent on many factors such as message size and process count.

We also tried to query the directory using CUDA asynchronous copy operations. Though this approach was feasible, it had high detrimental effect on the performance. The performance slowdown is basically due to the high number of asynchronous copy calls issued by the querying processes. These calls have to be synchronized at the beginning of each MPI_Allreduce invocation. Synchronization calls are costly, as they require waiting on all previously issued copies on the directory to complete. Avoiding synchronization calls can result in accessing stale data on the directory, which were stored in the previous invocation of MPI_Allreduce. This can ultimately result in inaccurate directory checking.

We also evaluated the effect of Hyper-Q feature in our designs. Hyper-Q allows multiple CPU threads or processes to launch work simultaneously on a single GPU. However, Hyper-Q (in CUDA 5.5) does not support CUDA IPC calls. In this regard, we modified our shared-buffer approach such that it does not use the CUDA IPC calls. To do so, we kept both directory and the shared buffers on the host memory. The host shared buffer was used to stage the data for GPU-to-GPU communication. The performance results of this approach with Hyper-Q enabled compared to Hyper-Q disabled showed only marginal improvement on large message sizes. The overhead of this approach is mainly due to the high cost of the data copy between host and GPU, and vice versa; on top of that, Hyper-Q works efficiently when processes only share computational resources of a single GPU (with no or few copies between the host and GPU). The reason lies in the fact that such copies are limited by the number of CPU and GPU memory controllers; thus, Hyper-Q does not parallelize GPU memory operations the way it does on computing operations.

Taking everything into consideration, with the current limitation on CUDA IPC, the best way to exploit this feature in our designs is to allocate the directory on the host main memory, while keeping the shared buffer on the GPU.

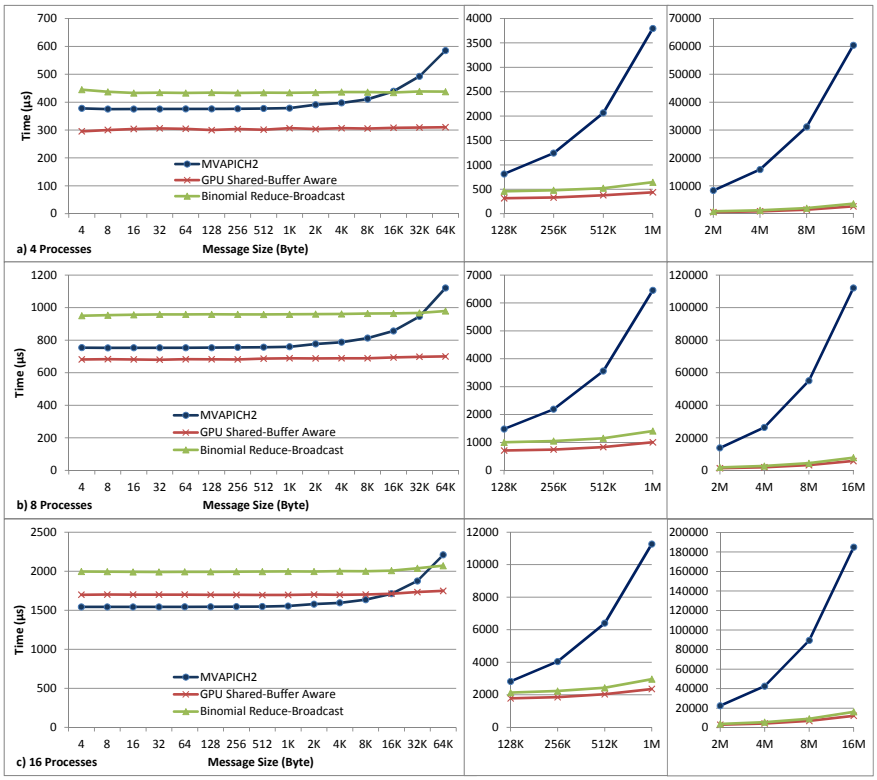

Figure 2: MVAPICH2 vs. GPU shared-buffer aware vs. Binomial reduce-broadcast MPI_Allreduce.

4. EXPERIMENTAL RESULTS

Our experimental system is a 16-core node equipped with an NVIDIA Kepler K20M GPU. The node has a dual socket Intel XEON E5-2650 clocked at 2.0 GHz, a 64 GB of memory, and is running a 64-bit Red Hat Enterprise Linux 6.2 and CUDA Toolkit 5.5. Our experimental results evaluate the proposed designs against MVAPICH2-1.9, using the OSU Micro-Benchmark 4.3 configured to support GPUs [2].

We present the results with various message sizes (4B - 16MB) on 4, 8, and 16 processes, in Fig. 2. Our results in different runs are consistent as ordered communications and computations in both designs prevent round-off errors.

According to Fig. 2, the GPU shared-buffer aware approach beats MVAPICH2 for all message sizes on 4 and 8 processes, while on 16 processes the benefit of the approach starts at message sizes greater than 16KB. The proposed designs show up to 22 times speedup over MVAPICH2. This is mainly due to the high cost of data staging in host memory as well as CPU-based reductions, used in the conventional design of MPI_Allreduce in MVAPICH2, compared to the lower cost of IPC copies and in-GPU reductions in our designs, especially for larger messages.

Clearly, the GPU shared-buffer aware approach beats the GPU-aware Binomial reduce-broadcast approach in all test cases. In all three cases, our results show a consistent behavior up to 128KB message sizes. The message size in this range has little effect on IPC copies and in-GPU reductions; however, as the message size increases, the reduction time and IPC time increase more rapidly. On the other hand, the costs of directory checking and issuing events are independent of message size.

As we increase the number of processes, the number of IPC calls also increases. The time for IPC calls mainly determines the total execution time of our approaches. These IPC calls cannot overlap, thus increasing the number of processes increases the total execution time, accordingly.

Initialization Overhead: Recall that in our designs, both of the host and GPU shared buffers are created during `MPI_Init()` and freed during `MPI_Finalize()`. For an application to benefit from the proposed designs, the one-time cost of this initialization overhead must be amortized. Fig. 3 shows the number of MPI_Allreduce calls required to compensate this initialization overhead in the GPU shared-buffer aware approach. As can be seen, for large message sizes, a few MPI_Allreduce calls (only a single call, in most cases) are just required to compensate for this overhead. For smaller messages, more calls are required. Note that, for the 16-process case, we have not shown the results for messages up to 16KB, as in this case the benefit of our approach starts at messages greater than 16KB (Fig 2.c).

5. RELATED WORK

There is a large body of literature showing the benefit of intranode communications in MPI hierarchical collectives in the CPU domain [4, 6]. Graham and Shipman showed the benefit of using on-node shared memory for various intranode MPI collectives [4]. Li et al. improved the performance of MPI collectives on clusters of NUMA nodes [6]; they proposed a multithreading-based MPI_Allreduce, which leverages shared memory for data transfers. While these studies

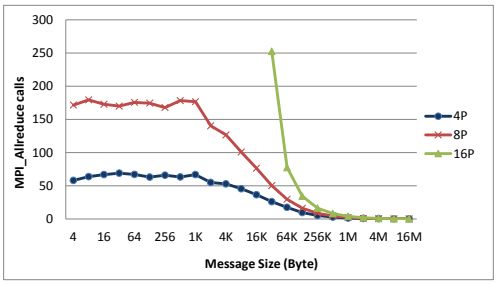

Figure 3: Number of MPI_Allreduce calls required to compensate for the initialization overhead.

consider intranode collectives in the CPU domain, our work targets intranode collectives in the GPU domain.

Some researchers have studied internode collective operations in GPU clusters [14, 11]. Singh et al. optimized internode MPI_Alltoall by basically overlapping network communication with the data movement between the CPU and the GPU [14]. Potluri et al. proposed various designs for internode MPI communications between GPU memories using GPUDirect RDMA [11]. The authors, as a part of their work, showed the benefit of GPUDirect on some MPI collectives. Their work targets collective communications among GPUs on different nodes and does not address intranode collective operations. Our work, on the other hand, targets intranode collectives and utilizes CUDA IPC feature.

The benefit of using CUDA IPC feature in intranode communications is studied in [12, 5]. However, this feature is only explored for one-sided and two-sided communications. Our paper, on the other hand, evaluates the benefit of this feature for intranode MPI_Allreduce collective.

6. CONCLUSION AND FUTURE WORK

Improving the performance of MPI_Allreduce is highly desirable. We investigated the efficient design and implementation of intranode MPI_Allreduce in GPU clusters. We proposed two design alternatives: a GPU shared-buffer aware, and a Binomial reduce-broadcast MPI_Allreduce. Both proposals use IPC copies and in-GPU reductions, along with GPU and host shared-buffers to achieve performance.

We discussed that the GPU shared-buffer aware approach is the best way to perform intranode MPI_Allreduce with the current restrictions on CUDA IPC. The results show that the GPU shared-buffer aware approach provides an improvement over the Binomial reduce-broadcast algorithm in all test cases. Both designs provide significant speedup over MVAPICH2 by up to 22 and 16 times, respectively.

As for future work, we first want to evaluate our designs on a larger multi-socket testbed. We would also like to extend our work to across the cluster by considering the internode stage of the hierarchical approach. Our proposed techniques, with minor changes, can also be applied to other collective operations; thus, we intend to evaluate our techniques as a general framework for MPI collective operations. NVIDIA in its very recent release of CUDA (CUDA 6.0) apparently supports CUDA IPC to work with Hyper-Q. As a part of future work, we are interested to evaluate our proposed techniques and come up with new designs for MPI_Allreduce and other collectives with this new release.

7. ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant #RGPIN/238964-2011, Canada Foundation for Innovation, and Ontario Innovation Trust Grant #7154.

8. REFERENCES

[1] A. M. Aji, L. S. Panwar, F. Ji, M. Chabbi, K. Murthy, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey, X. Ma, and R. Thakur. On the Efficacy of GPU-integrated MPI for Scientific Applications. In *HPDC'13*, pages 191–202, 2013.

[2] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda. OMB-GPU: A Micro-benchmark Suite for Evaluating MPI Libraries on GPU Clusters. In *EuroMPI'12*, pages 110–120, 2012.

[3] CUDA, http://www.nvidia.ca/object/cuda_home_new.html (Last accessed 5/02/2014).

[4] R. L. Graham and G. Shipman. MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In *PVM/MPI'2008*, pages 130–140. 2008.

[5] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, R. Thakur, W.-c. Feng, and X. Ma. DMA-Assisted, Intranode Communication in GPU Accelerated Systems. In *HPCC'12*, pages 461–468, 2012.

[6] S. Li, T. Hoefler, and M. Snir. NUMA-Aware Shared Memory Collective Communication for MPI. In *HPDC'13*, pages 85–96, 2013.

[7] MPI-3, http://www.mpi-forum.org/docs/mpi-3.0/ (Last accessed 4/28/2014).

[8] MVAPICH, http://mvapich.cse.ohio-state.edu (Last accessed 4/30/2014).

[9] NVIDIA Corporation: NVIDIA CUDA C Programming Guide Version 4.1, 2011.

[10] A. J. Pena and S. R. Alam. Evaluation of Inter- and Intra-node Data Transfer Efficiencies between GPU Devices and their Impact on Scalable Applications. In *CCGrid'13*, pages 144–151, 2013.

[11] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. Efficient Inter-node MPI Communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *ICPP'13*, pages 80–89, 2013.

[12] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda. Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication. In *IPDPSW'12*, pages 1848–1857, 2012.

[13] R. Rabenseifner. Automatic Profiling of MPI Applications with Hardware Performance Counters. In *MPIDC'99*, pages 77–85, 1999.

[14] A. K. Singh, S. Potluri, H. Wang, K. Kandalla, S. Sur, and D. K. Panda. MPI Alltoall Personalized Exchange on GPGPU Clusters: Design Alternatives and Benefit. In *Cluster'11*, pages 420–427, 2011.

[15] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective Communication Operations in MPICH. *IJHPCA'05*, 19(1):49–66, 2005.

[16] The TOP500 June 2014 List, http://www.top500.org/lists/2014/06/.