Enabling Efficient Use of MPI and PGAS Programming Models on Heterogeneous Clusters with High Performance Interconnects

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Sreeram Potluri, M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2014

Dissertation Committee:

Dhabaleswar K. Panda, Advisor
Ponnuswamy Sadayappan
Radu Teodorescu
Karen Tomko
© Copyright by
Sreeram Potluri
2014
Abstract

Accelerators (such as NVIDIA GPUs) and coprocessors (such as Intel MIC/Xeon Phi) are fueling the growth of next-generation ultra-scale systems with high compute density and high performance per watt. However, they render these systems heterogeneous by introducing multiple levels of parallelism and varying computation/communication costs at each level. Application developers also use a hierarchy of programming models to extract maximum performance from these heterogeneous systems. They use models like CUDA, OpenCL, LEO+OpenMP, and others on an accelerator or a coprocessor while using higher level programming models like an MPI or a PGAS model across a cluster. Multiple programming models, their runtimes, and varying performance of communication at different levels of the system hierarchy have limited applications from the achieving peak performance on these systems. For example, in MPI and OpenSHMEM applications that run on GPU clusters, data is transferred between GPU and CPU using CUDA while it is exchanged between MPI processes running on different nodes using MPI or OpenSHMEM. This two-stage process for data movement introduces inefficiencies and hence limits performance. Communication in applications running on clusters with Intel MIC can happen over a myriad of channels depending on where the application processes are running: intra-MIC, intra-host, MIC-Host and MIC-MIC. Each of these channels has different performance characteristics. Runtimes have to be re-designed to optimize communication in such scenarios, while hiding system complexity from the user.
Computation and communication overlap has been a critical requirement for applications, to achieve peak performance on large-scale systems. Communication overheads have a magnified impact on heterogeneous clusters due to their higher compute density and hence, a higher wastage in compute power. Modern interconnects like InfiniBand, with their Remote DMA capabilities, enable asynchronous progress of communication, freeing up the cores to do useful computation. MPI and PGAS models offer light-weight, one-sided communication primitives that minimize process synchronization overheads and enable better computation and communication overlap. However, the design of one-sided communication on heterogeneous clusters is not well studied. Further, there has been limited literature to guide scientists in taking advantage of the one-sided communication semantics on high-end applications, more so on heterogeneous architectures.

This dissertation has targeted several of these challenges for programming on GPU and Intel MIC clusters. Our work with MVAPICH2-GPU enabled the use of MPI in a unified manner, for communication from host and GPU device memories. It takes advantage of unified virtual addressing (UVA) provided by CUDA. We proposed designs in the MVAPICH2-GPU runtime to significantly improve the performance of internode and intranode GPU-GPU communication by pipelining and overlapping memory, PCIe and network transfers. We take advantage of CUDA features, such as IPC, GPUDirect RDMA, and CUDA kernels to further reduce communication overheads. MVAPICH2-GPU improves programmability by removing the need for developers to use CUDA and MPI for GPU-GPU communication, while improving performance through runtime-level optimizations that are transparent to the user. We have shown up to 69% and 45% improvement in point-to-point latency for data movement for 4Byte and 4MB messages, respectively. Likewise, the solutions improve the bandwidth by 2x and 56% for 4KByte and 64 KByte
messages, respectively. Our work have been released as part of MVAPICH2 packages, making it the first MPI library to support direct GPU-GPU communication. It is currently deployed and used on several large GPU clusters across the world, including Tsubame 2.0 and Keeneland. We proposed novel extensions to the OpenSHMEM PGAS model that enable unified communication from host and GPU memories. We present designs for optimized internode and intranode one-sided communication on GPU clusters, using asynchronous threads and DMA-based techniques. The proposed extensions, coupled with an efficient runtime, improve the latency of 4 Byte \texttt{shmem\_getmem} latency by 90%, 40%, and 17%, for intra-IOH, inter-IOH, and inter-node GPU configurations with CUDA, respectively. They improve the performance of Stencil2D and BFS kernels by 65% and 12% on clusters of 192 and 96 GPUs, respectively.

Through MVAPICH2-MIC, we proposed designs for an efficient MPI runtime on clusters with Intel Xeon Phi coprocessors. These designs improve performance of Intra-MIC, Intra-Node and Inter-Node communication on various cluster configurations, while hiding the system complexity from the user. Our designs take advantage of SCIF, Intel’s low-level communication API, in addition to standard communication channels like shared memory and IB verbs, to offer substantial performance gains in performance of the MVAPICH2 MPI library. PRISM, a proxy-based multi-channel design in MVAPICH2-MIC allows applications to overcome the performance bottlenecks imposed by state-of-the-art processor architectures and extract the full compute potential of the MIC coprocessors. The proposed designs deliver up to 70% improvement in the point-to-point latency and more than 6x improvement in peak uni-directional bandwidth from Xeon Phi to the Host. Using PRISM, we improve inter-node latency between MICs by up to 65% and bandwidth by up to 5 times.
PRISM improves the performance of MPI_Alltoall operation by up to 65%, with 256 processes. It improves the performance of 3D Stencil communication kernel and P3DFFT library by 56% and 22% with 1024 and 512 processes, respectively.

We have shown the potential benefits of using MPI one-sided communication semantics for overlapping computation and communication, in a real-world seismic modeling application, AWP-ODC. We have shown a 12% improvement in overall application performance on 4,096 cores. This effort was also part of the application’s entry as a Gordon Bell finalist at SC’2010. We demonstrated the potential performance benefits of using one-sided communication semantics on GPU clusters. We presented an efficient design for MPI-3 RMA model on NVIDIA GPU clusters with GPUDirect RDMA and proposed minor extensions to the model that can further reduce synchronization overheads. The proposed extension to the RMA model enables an inter-node ping-pong latency of 2.78 usec between GPUs - a 60% improvement over latency offered by send/recv operations. One-sided communication provides 2x the message rate achieved using MPI Send/Recv operations. One-sided semantics improve the latency of a 3DStencil communication kernel - by up to 27%.
To my family, friends, and mentors.
Acknowledgments

This work was made possible through the love and support of several people who stood by me, through the many years of my doctoral program and all through my life leading to it. I would like to take this opportunity to thank all of them.

My family - my father, Dr. Gangadhara Rao, who has always given me inspiration and complete freedom to go after my dreams; my mother, Dr. Sarada, who has given me the love and support to venture forth; my brothers and sisters, Vamsi, Kavitha, Swapna and Sriram, who have stood by me at all times and my in-laws, Malleswara Rao and Vijaya Lakshmi, for giving me the most invaluable gift of my life.

My parents away from home - my brother-in-law, Anil Thumati, who had always treated me as his brother and friend, giving me the guidance and love only the best of brothers can give; my sister, Swapna Thumati, whom I have always admire and who has always been there for me, especially when the going was tough; my darling nieces, Charvi and Krishi, who have made the years in Columbus the most beautiful years of my life.

My wife, Pujitha, for her love, support and understanding. I admire and respect her for the many qualities she possesses, particularly the great courage and presence of mind she has at the most trying of times. She is a joy to be around.

My advisor, Dr. Dhabaleswar K. Panda for his guidance and support throughout my doctoral program. I have been able to grow, both personally and professionally, through
my association with him. Even after knowing him for six years, I am amazed by the energy and commitment he has towards research.

My collaborators - I would like to thank all my collaborators: Karen Tomko, Sayantan Sur and Devendar Bureddy, who have been instrumental in several publications that constitute this thesis.

My friends - Naveen Kanthamneni and Dheeraj Pulluri, who have seen and supported me through the highs and lows of my Ph.D. life. I am very happy to have met and become friends with Hari Subramoni, Krishna Kandalla, Raghunath Rajachandrasekhar, Khaled Hamidouche, Akshay Venkatesh, Jithin Jose, Jonathan Perkins and Mark Arnold. This work would remain incomplete without their support and contribution. They have given me memories that I will cherish for the rest of my life.

I would also like to thank all my colleagues, who have helped me in one way or another throughout my graduate studies.

Above all, I would like to thank the Almighty for making all this possible.
Vita

2001-2005 ........................................B.Tech., Computer Science and Engineering, Jawaharlal Nehru Technological University, India

2006-2008 ........................................Software Engineer, Infosys Technologies Ltd., India

2008-2013 ........................................M.S., Computer Science and Engineering, The Ohio State University, U.S.A

2008-Present ......................................Ph.D., Computer Science and Engineering, The Ohio State University, U.S.A

2009-Present ......................................Graduate Research Associate, The Ohio State University, U.S.A

2010 ....................................................Summer Research Intern, Argonne National Laboratory, U.S.A.

Publications


A. Venkatesh, S. Potluri, R. Rajachandrasekar, M. Luo, K. Hamidouche, and D. K. Panda, High Performance Alltoall and Allgather designs for InfiniBand MIC Clusters *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2014(Accepted for publication).


J. Jose, S. Potluri, K. Tomko and D. K. Panda, Designing Scalable Graph500 Benchmark with Hybrid MPI+OpenSHMEM *Programming Models International Supercomputing Conference (ISC)*, June 2013.


Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>vi</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vii</td>
</tr>
<tr>
<td>Vita</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xvi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvii</td>
</tr>
<tr>
<td>Chapter 1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Statement</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Research Framework</td>
<td>7</td>
</tr>
<tr>
<td>Chapter 2. Background</td>
<td>13</td>
</tr>
<tr>
<td>2.1 InfiniBand</td>
<td>13</td>
</tr>
<tr>
<td>2.1.1 Communication Model and Transports</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2 Communication Semantics</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Programming Models for High Performance Computing</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1 MPI</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 PGAS Models and OpenSHMEM</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Communication on Clusters with GPUs</td>
<td>21</td>
</tr>
<tr>
<td>2.4 Communication on Clusters with Intel MIC based Coprocessors</td>
<td>25</td>
</tr>
</tbody>
</table>
Chapter 3. MVAPICH2-GPU: Optimizing Communication on GPU Clusters using MPI

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Unified Interface for Data Movement</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Inter-node Communication</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Pipelined Communication</td>
<td>31</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Designs using GPUDirect RDMA</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Intra-node Communication on Multi-GPU Clusters</td>
<td>41</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Designs using CUDA IPC</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>Experimental Evaluation</td>
<td>45</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Internode Communication</td>
<td>45</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Intranode Communication</td>
<td>55</td>
</tr>
<tr>
<td>3.5</td>
<td>Related Work</td>
<td>61</td>
</tr>
<tr>
<td>3.6</td>
<td>Summary</td>
<td>63</td>
</tr>
</tbody>
</table>

Chapter 4. Extending OpenSHMEM for GPU Computing

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Limitations of Existing Alternatives</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>Extending the OpenSHMEM Memory Model</td>
<td>69</td>
</tr>
<tr>
<td>4.3</td>
<td>Symmetric Heap and Address Translation</td>
<td>72</td>
</tr>
<tr>
<td>4.4</td>
<td>Designs for an Efficient Communication Runtime</td>
<td>74</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Intra-node Communication</td>
<td>74</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Inter-node Communication</td>
<td>76</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Atomics and Synchronization</td>
<td>78</td>
</tr>
<tr>
<td>4.5</td>
<td>Redesigning Application Kernels</td>
<td>79</td>
</tr>
<tr>
<td>4.6</td>
<td>Experimental Evaluation</td>
<td>80</td>
</tr>
<tr>
<td>4.7</td>
<td>Related Work</td>
<td>86</td>
</tr>
<tr>
<td>4.8</td>
<td>Summary</td>
<td>89</td>
</tr>
</tbody>
</table>

Chapter 5. MVAPICH2-MIC: Efficient MPI Communication on Intel MIC Clusters

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Intra-MIC Communication</td>
<td>90</td>
</tr>
<tr>
<td>5.2</td>
<td>Intra-node Communication Between MIC and Host</td>
<td>92</td>
</tr>
<tr>
<td>5.3</td>
<td>PRISM for Internode Communication</td>
<td>94</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Design Considerations</td>
<td>95</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Direct IB</td>
<td>97</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Passive Proxy</td>
<td>97</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Active Proxy</td>
<td>99</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Discussion</td>
<td>101</td>
</tr>
<tr>
<td>5.4</td>
<td>Experimental Evaluation</td>
<td>102</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Intra-node Communication</td>
<td>102</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Peer-to-Peer Performance on Intel Architectures and Percentage of Bandwidth offered by FDR IB Adapter (6397 MB/s)</td>
<td>25</td>
</tr>
<tr>
<td>6.1 Latencies (half round trip) of IB Send/Recv and MPI Send/Recv latencies for data movement between Hosts and GPUs (in usec)</td>
<td>142</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Data Movement using OpenSHMEM on GPU Clusters</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Different Paths of Communication in a Cluster with Intel Xeon Phi Copro-</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>cessors</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>Research Framework</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>MPI-2 One-Sided Model and Communication using Post-Wait/Start-Complete</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Semantics</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Communication Channels among MICs and Hosts</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Data Flow from GPU memory to remote GPU memory</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Pseudocode for existing approaches for data movement on GPU clusters</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Pseudocode using MVAPICH2-GPU (Good Performance and Good Productivity)</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>MVAPICH2-GPU with GPU-Direct</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Comparison and Cross-over between IB RDMA Transfer and CUDA Memory Copy</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Proposed GDR based Inter-node GPU-to-GPU Pipelined Data Movement in MVAPICH2</td>
<td>40</td>
</tr>
</tbody>
</table>
3.7 Steps involved in an intra-node eager GPU-to-GPU communication from Process A to Process B: (1) Process A does IPC copy from source device buffer to eager device buffers at Process B, (2) Process A calls \texttt{cuEventRecord} the IPC event, (3) Process A copies the header into eager host buffers at Process B, (4) Process B detects the header and calls \texttt{cuStreamWaitEvent} on the IPC event, and (5) Process B issues a copy from its eager device buffers to the destination eager buffer.

3.8 Two-sided Latency Performance using Pipelined Designs

3.9 Collectives Latency Performance using Pipelined Designs

3.10 Comparison of Latency Performance Using the Existing Host-based Pipelining and the Proposed GDR-based Designs on Sandy Bridge

3.11 Comparison of Bandwidth Performance Using the Existing Host-based Pipelining and the Proposed GDR-based Designs on Sandy Bridge

3.12 Comparison of Bi-bandwidth Performance Using the Existing Host-based Pipelining and the Proposed GDR-based Designs on Sandy Bridge

3.13 Impact of GDR-based Designs on MPI\_Gather communication performance

3.14 Impact on Performance of Lattice Boltzmann Method code

3.15 Impact on Performance of AWP-ODC code

3.16 Performance of CUDA IPC

3.17 Latency Performance of Two-sided Communication

3.18 Bandwidth Performance of Two-sided Communication

3.19 Per-step latency in GPULBM. Two processes on an 8-core Intel Westmere node with two NVIDIA C2075 GPUs are used.

4.1 Programming with current and enhanced OpenSHMEM Models

4.2 Existing Heap Selection Based Designs

4.3 Symmetric Heap Model: Mapping device buffer objects
4.4 Extensions to OpenSHMEM for Symmetric Map .................................. 70
4.5 Intra-Node communication using CUDA: Intra-IOH and Inter-IOH Configurations ......................................................... 75
4.6 Inter-node communication using CUDA or OpenCL ............................ 77
4.7 Intranode \texttt{shmem\_putmem} Performance with CUDA .................... 82
4.8 Intranode \texttt{shmem\_getmem} Performance with CUDA .................... 82
4.9 Internode \texttt{shmem\_getmem} Performance with CUDA .................... 84
4.10 Internode \texttt{shmem\_putmem} Performance with OpenCL ................. 85
4.11 OpenSHMEM Atomics Performance with CUDA ............................... 86
4.12 SHOC Application Benchmark Performance with CUDA on up to 192 GPUs (3 GPUs/node) ................................................. 87
5.1 Communication Channels within a MIC in MVAPICH2 ....................... 92
5.2 Communication Between Host and MIC ........................................... 94
5.3 Comparing Design Alternatives for Inter-node MIC-MIC Communication using IB .......................................................... 96
5.4 Performance of Intra-MIC Communication ....................................... 104
5.5 Performance of Communication between MIC and Host .................... 105
5.6 Uni-directional Bandwidth between MIC and Host Communication ........ 106
5.7 Performance of Collective Operations with 8 Processes on Host + 8 Processes on MIC ...................................................... 107
5.8 Performance of 3D Stencil Communication Kernel and P3DFFT Application110
5.9 Latency performance of inter-node MIC-MIC communication ............ 111
5.10 Bandwidth performance of inter-node MIC-MIC communication  

5.11 Latency performance of inter-node MIC-Host communication  

5.12 Bandwidth performance of inter-node MIC-Host communication  

5.13 Performance of Alltoall communication with 16 MIC coprocessors, up to 256 processes  

5.14 Performance of MPI_Gather communication with 32 MIC coprocessors, up to 512 processes  

5.15 Performance of MPI_Allgather communication with 32 MIC coprocessors, up to 256 processes  

5.16 Performance of 3DStencil Communication Kernel with 32 MIC coprocessors, up to 512 processes  

5.17 Performance of 3DStencil Communication Kernel with 32 MIC coprocessors and hosts, up to 1,024 processes  

5.18 Performance of P3DFFT in Pure MIC configuration  

5.19 Performance characteristics of the proposed designs  

6.1 Data Transfer Patterns in AWP-ODC  

6.2 AWP-ODC Application Pseudo-Code  

6.3 Analysis of AWP-ODC run-time  

6.4 Velocity Exchange with Reduced Synchronization using Non-blocking Two-sided Semantics  

6.5 Velocity Exchange with Reduced Synchronization using Post-Wait/Start-Complete  

6.6 Comparison of various design enhancements (Left) main loop execution times (Right) percentage improvement  

6.7 Component Analysis of the Original Main Loop Time
6.8 Communication Completion and Notification in Different Communication and Synchronization Mechanisms in MPI-3. (App: Application, RT: Runtime) .................................................. 161

6.9 Implementation alternatives for the proposed extension ........................................ 162

6.10 Proxy-based Framework to Efficiently Support RMA Model on GPU clusters 162

6.11 Comparison of Latency and Bandwidth Performance of Send/Recv and RMA Primitives ................................................................. 163

6.12 Performance of 3D Stencil Communication using Send/Recv and One-sided Primitives - Contiguous Data Transfers ................................. 164

6.13 Performance of Alltoall Communication using Send/Recv and One-sided Primitives ................................................................. 165

6.14 Performance of 3D Stencil Communication using Send/Recv and One-sided Primitives - Non-contiguous Data Transfers ......................... 165

6.15 Impact of Proxy-based Framework to Handle Large Message RMA Communication ................................................................. 166

6.16 Latency Performance and Passive Progress of Get with Passive Synchronization ................................................................. 167

6.17 Asynchronous Progress of Get with Passive Synchronization ................................. 168
Chapter 1: Introduction

Accelerators (such as NVIDIA GPUs) and coprocessors (such as Intel Many Integrated Core/Xeon Phi) are fueling the growth of next-generation ultra-scale systems that have a high compute density and a high performance per watt. This is evident in the recent Top500 list [16] (November 2013) with 25 of the top 100 systems using either accelerators or coprocessors to boost their compute power. This includes Tianhe-2, Titan, Piz Daint, and Stampede from the top 10. Networking technologies like InfiniBand (IB) [40] have also evolved rapidly over the years, offering low latency and high bandwidth to address the increasing communication demands of these high-density systems. The latest generation of InfiniBand adapter from Mellanox Technologies offers sub-micro-second latency and more than 100 Gb/sec in bandwidth. The many-core technologies and high-performance networks are expected to be part of the solution for exascale computing.

Message Passing Interface (MPI) has become the de-facto standard for developing parallel applications in the scientific computing domain over the last two decades [18]. Portability and availability of high-performance implementations on modern architectures have been the key reasons for its popularity. MPI also offers flexibility for designing communication with different kinds of semantics including two-sided and one-sided semantics. Partitioned Global Address Space (PGAS) Models, on the other hand, have evolved as an attractive alternative to MPI. They provide simple shared memory abstractions while
exposing data locality information to application developers [66, 93]. This improves programmability while also allowing users to move data efficiently. Unified Parallel C (UPC) and OpenSHMEM are popular examples of compiler-based and library-based PGAS programming models. Newer programming frameworks like CUDA, OpenCL, and others have evolved that enable applications to harness the massive parallelism provided by GPUs and coprocessors [1, 64].

Many-core architectures like GPUs and MIC-based coprocessors have caused systems to become heterogeneous by introducing multiple levels of parallelism and varying computation/communication costs at each level. Application developers also use a hierarchy of programming models to extract maximum performance from these heterogeneous systems. Models such as CUDA, OpenCL, LEO (Intel Language Extensions for Offload)+OpenMP, and others are used to express parallelism across accelerator or coprocessor cores while higher level programming models such as MPI, OpenSHMEM, or UPC are used to express parallelism across a cluster. The presence of multiple programming models, their runtimes and the varying communication performance at different levels of the system hierarchy has hindered applications from achieving peak performance on these systems.

For example, in an MPI or OpenSHMEM application with processes moving data between GPUs, data is transferred from local GPU to local host using CUDA or OpenCL, exchanged between processes using MPI or OpenSHMEM and finally transferred from remote host to remote GPU again using CUDA or OpenCL. This two-level process for data movement introduces inefficiencies and limits performance that can be extracted from underlying hardware. In the case of OpenSHMEM, this removes the benefits of one-sided communication by requiring synchronization between processes. This is depicted in Figure 1.1(a). Further, modern system architectures allow GPU nodes to be configured in
several ways with one or more GPUs and network adapters per node. GPU programming frameworks like CUDA also offer several low-level features to optimize data movement on these GPU clusters. The usability and efficiency of these techniques depends heavily on the way GPUs are configured. Intricacies like these make it even harder for application developers to optimize communication on GPU clusters. It is important that data movement is optimized by taking advantage of the various features offered by the underlying hardware, in a way that is transparent to the user. This raises the following broad challenges: “Can popular programming models like MPI and OpenSHMEM be extended so that they can be used effectively for data movement on GPU clusters? Can their runtimes deliver optimal communication performance by taking advantage of the features offered by the system hardware, in a way that is transparent to the application developer?” A simplified communication code using the OpenSHMEM model envisioned for data movement directly from device memory is depicted in Figure 1.1(b).

Intel MIC architecture, with its x86 compatibility, presents an attractive alternative to GPUs, for enhancing performance of HPC systems [3]. It supports many of the popular programming models including MPI, OpenMP, Thread Building Blocks, and others that are used on standard multi-core architectures. This dramatically reduces the effort required for porting applications developed for multi-core systems, onto MIC-based coprocessors. MIC architecture provides a highly flexibility usage model for application developers. A MIC-based coprocessor like the Xeon Phi can be used in a many-core hosted mode to run applications like on any standard multi-core node or in a offload mode where applications can offload compute like to a GPU accelerator. It also offers a symmetric mode where MPI processes can be launched on both the coprocessor and the host. As shown in Figure 1.2, applications running in these different modes can have up to nine paths of communication
if(pe == 0) {
    cudaMemcpy(s_buf, s_device, size, cudaMemcpyDeviceToHost);
    shm_putmem(s_buf, r_buf, size, 1);
    shm_barrier(...)
} else if (pe == 1) {
    shm_barrier(...)
    cudaMemcpy(r_device, r_buf, size, cudaMemcpyHostToDevice);
}

(a) Existing Model

if(pe == 0) {
    shm_putmem(s_device, r_device, size, 1);
} else if (pe == 1) {
    <no calls required>
}

(b) Proposed Model

Figure 1.1: Data Movement using OpenSHMEM on GPU Clusters
depending on where the application processes are running. Each of these levels of communication can be implemented using different channels offered by the underlying system. Further, each of these channels has different performance characteristics. These make an optimal communication design a hard task. “How can existing communication runtimes be re-designed to efficiently support different levels of communication on a MIC cluster while making it easier for applications to extract maximum performance on the clusters with the Intel MIC-based coprocessors?” It is imperative that this is addressed for the efficient use and hence, the wider acceptance of the new MIC technology.

![Diagram](image)

Figure 1.2: Different Paths of Communication in a Cluster with Intel Xeon Phi Coprocessors

Computation and communication overlap has been a critical requirement for applications to achieve peak performance on large scale systems. Communication overheads have a magnified impact on heterogeneous clusters due to their higher compute density and hence a larger waste in compute power. Modern interconnects such as InfiniBand, with
their Remote DMA capabilities, enable asynchronous progress of communication, thereby freeing up the cores to do useful computation. MPI and PGAS models offer light weight one-sided communication primitives that minimize process synchronization overheads and enable better computation and communication overlap [7]. However, the design of one-sided communication on heterogeneous clusters is not well studied. Further, there is limited literature available to guide scientists in taking advantage of the one-sided communication semantics on high-end applications, and even more so on heterogeneous architectures. “How can one-sided communication be efficiently designed on heterogeneous clusters? How can applications take advantage of one-sided semantics to achieve better overlap?” These can enable applications to take maximum advantage of modern high-performance interconnects and minimize communication overheads on heterogeneous clusters.

1.1 Problem Statement

It is critical for the issues outlined above to be addressed in order to ensure applications make efficient use of the compute power and network resources offered by modern heterogeneous clusters. Developers should have an easy-to-use way to move data between the accelerators or co-processors on these clusters, to ensure high productivity. The communication runtimes should be able to efficiently take advantage of all the features offered by the underlying hardware and software, to ensure maximum performance. Further, applications should be designed using communication semantics that help deliver close to network-level performance and also allow communication overheads to be overlapped with computation. In this thesis, we target these issues on clusters with GPUs and Intel MIC-based coprocessors. To summarize, it addresses the following broad challenges:
1. Can MPI and PGAS models be extended and used to improve the efficiency of communication on heterogeneous clusters with GPUs?

2. How can communication runtimes be designed to take advantage of the various features offered by GPU programming frameworks while addressing the limitations posed at various levels of the system?

3. Can communication runtimes be designed to support efficient communication along all of the paths on heterogeneous clusters with Intel MIC coprocessors?

4. What kind of benefits can be achieved using heterogeneous architecture-aware MPI and PGAS libraries in scientific kernels and applications?

5. Can high-end applications be redesigned using one-sided semantics to increase communication and computation overlap and achieve better performance?

6. How can we design one-sided communication primitives on heterogeneous architectures and what is their impact on performance?

1.2 Research Framework

Figure 1.3 depicts the research framework that we present to address the challenges highlighted above. We discuss how we use the framework to addresses each of the challenges in detail.

1. Can MPI and PGAS models be extended and used to improve the efficiency of communication on heterogeneous clusters with GPUs?

The paradigm of using CUDA or OpenCL for data movement between GPUs and host memory while using models like MPI or PGAS for data movement makes it
hard to optimize communication in applications running on GPU clusters. The application developers have to employ intricate designs like pipelining using the two models to reduce communication overheads. These designs also may not lead to optimal performance because of the lack of interaction between the runtimes of the two models involved. This leads to low developer productivity and sub-optimal performance. In the research framework, we propose the idea of using a unified interface for data movement on GPU clusters. We enable MPI and OpenSHMEM PGAS models to be used for communication from GPU device memory, without requiring the user to make any CUDA or OpenCL calls. This simplifies the communication code
for the application developer while it enables the MPI and OpenSHMEM runtimes to optimize communication, in a way that is transparent to the user.

2. How can communication runtimes be designed to take advantage of the various features offered by GPU programming frameworks while addressing the limitations posed at various levels?

CUDA and OpenCL offer several features to optimized data movement from and into a GPU. For example both CUDA and OpenCL have a non-blocking data copy operations and non-blocking checks for their completion that can be used to overlap data movement between GPU and Host with data movement across the network or with computation. Further, CUDA offers Inter-Process Communication (IPC) that delivers superior performance for data movement from one GPU to another but can only be used between GPUs that are connected in certain intra-node configurations. Also, techniques like GPUDirect RDMA offer low-latency and high-bandwidth inter-node communication for certain range of message sizes and configurations while providing sub-optimal performance in other scenarios due to system limitations as highlighted in Section 2.3. It is hard for an application developer to detect and consider all these options while designing communication. In this dissertation, we present high-performance MPI and OpenSHMEM runtimes that use techniques like pipelining for host-staged communication and take advantage of advanced technologies like CUDA IPC and GPUDirect RDMA where applicable. They do this in a transparent manner where the application developer uses MPI or OpenSHMEM calls to initiate communication from GPU buffers while the runtime takes care of efficiently implementing the actual transfer.
3. Can communication runtimes be designed to support efficient communication along all the channels on heterogeneous clusters with Intel Xeon Phi coprocessors?

The Intel MIC architecture provides several ways in which applications can take advantage of the co-processors. For example, MPI applications can use the coprocessor as an accelerator to offload computation, as a multi-core node with MPI processes directly running on it, or in a symmetric mode where processes run on both host and coprocessor. Symmetric mode provides the maximum flexibility in taking advantage of the resources on the host and the coprocessors. However, this involves communication along different paths as explained in Section 1. The MPSS software stack running on the MIC provides various channels for communication depending on where the processes are running. For example, intra-MIC communication can be designed using POSIX shared memory, a low-level interface called SCIF or network loopback when a network adapter is available. We propose designs for an MPI runtime that considers the different channels while implementing communication along each of the different paths to extract maximum communication performance. It supports efficient communication along all of the paths to ensure that applications can run efficiently on a cluster with Xeon Phis. We propose a proxy-based runtime that overcomes the bottlenecks found in PCIe transfers between the Xeon Phi and the network, on systems with state-of-the-art processor architectures.

4. What kind of benefits can be achieved using heterogeneous architecture-aware MPI and PGAS libraries in scientific kernels and applications?

Traditionally, applications had to use CUDA or OpenCL and a higher level programming model like MPI or PGAS to design communication on GPU clusters. The idea
of GPU-aware MPI and PGAS models changes the way communication is designed in applications. It enables developers to take advantage of features offered by MPI and PGAS models for direct communication from GPU memory. As part of the framework, we consider a diverse set of end-applications and kernels. They range from a stencil-based seismic modeling code with near-neighbor communication pattern to a graph traversal code with dynamic communication pattern. We discuss how these codes benefit from the models proposed for GPU clusters. We evaluate the impact of optimized runtimes designed for GPU and MIC clusters on the performance of these end-applications and kernels.

5. Can high-end applications be redesigned using one-sided semantics to increase communication and computation overlap and achieve better performance?

One-sided communication semantics in MPI and PGAS models separate data movement from synchronization between processes. They also provide non-blocking Put, Get primitives, that can be efficiently implemented over RDMA-like channels available on modern architectures, delivering close-to-peak performance. They allow applications to reduce synchronization overheads and to overlap communication with computation. There has been very little work that shows the efficient use of one-sided communication in end-applications. The existing work has primarily focused on applications with dynamic communication patterns and with bandwidth-critical communication phases. In this report, we consider a seismic modeling code with a near-neighbor communication pattern that is common in a wide range of applications. We demonstrate how such an application can be re-designed using MPI one-sided communication primitives to achieve optimal computation and communication overlap.
6. How can we design one-sided communication primitives on heterogeneous architectures and what is their impact on performance?

Heterogeneous architectures offer communication channels that have RDMA semantics. For example, newer versions of CUDA offer IPC, for direct data movement between GPUs connected on the same CPU socket and GPUDirect RDMA, which offers direct data movement between GPUs over high-performance networks like InfiniBand. One-sided semantics have shown to help achieve close-to-peak performance while using these communication channels, when compared to two-sided semantics. In this report, we present an efficient implementation of MPI one-sided semantics on GPU clusters using CUDA IPC and GPUDirect RDMA. We show how they can achieve better performance than two-sided semantics, using micro-benchmarks and communication kernels.
Chapter 2: Background

2.1 InfiniBand

InfiniBand (IB) is an industry standard switched fabric that is designed for interconnecting compute and I/O nodes in High-End Computing clusters [2]. It has emerged as the most-used internal systems interconnect in the Top 500 list of supercomputers. The list released in November 2013 reveals that more than 40% of the systems use IB.

2.1.1 Communication Model and Transports

Connection establishment and communication in IB is done using a set of primitives called Verbs. IB uses a queue based model. A process can queue up a set of instructions that the hardware executes. This facility is referred to as a Work Queue (WQ). Work queues are always created in pairs, called a Queue Pair (QP), one for send operations and one for receive operations. In general, the send work queue holds instructions that cause data to be transferred from one process’s memory to another process, and the receive work queue holds instructions about where to place data that is received. The completion of Work Queue Entries (WQEs) is reported through Completion Queues (CQ). IB provides both reliable and unreliable transport modes: Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC) and Unreliable Datagram (UD). However, only RC and UC
are required to be supported for IB implementations to be compliant with the standard. The RC transport is used in all the experiments presented in this report. As its name suggests, RC provides reliable communication and ensures ordering of messages between two endpoints. Memory involved in communication through IB should be registered with the IB network adapter. Registration is done using an IB verbs call which pins the corresponding pages in memory and returns local and remote registration keys (lkey and rkey). The keys are used in communication operations as described in the following section.

### 2.1.2 Communication Semantics

InfiniBand supports two types of communication semantics in RC transport: channel and memory semantics. In channel semantics, both the sender and receiver have to be involved to transfer data between them. The sender has to post a send work request entry (WQE) which is matched with a receive work request posted by the receiver. The buffer and the lkey are specified with the request. It is to be noted that the receive work request needs to be posted before the data transfer gets initiated at the sender. The receive buffer size should be equal or greater than the send buffer size. This restriction is prevalent in most high-performance networks like Myrinet [21], Quadrics [13] and others. This allows for zero-copy transfers but requires strict synchronization between the two processes. Higher level libraries avoid this synchronization by pre-posting receive requests with staging buffers. The data is copied into the actual receive buffer when the receiver posts the receive request. This allows the send request to proceed as soon as it is posted. There exists a trade-off between synchronization costs and additional copies. In memory semantics, Remote Direct Memory Access (RDMA) operations are used instead of send/receive operations. These RDMA operations are one-sided and do not require software involvement at the target. The
remote host does not have to issue any work request for the data transfer. The send work request includes address and lkey of the source buffer and address and rkey of the target buffer. Both RDMA Write (write to remote memory location) and RDMA Read (read from remote memory location) are supported in InfiniBand.

2.2 Programming Models for High Performance Computing

This thesis focuses on two of the most widely used programming paradigms in the High Performance Computing (HPC) domain: Message Passing Interface (MPI) and Partitioned Global Address Space (PGAS) Models.

2.2.1 MPI

Over the last two decades, MPI has become the de-facto standard for developing scientific applications in the HPC domain. Portability and availability of high-performance implementations on most modern architectures have been key factors in wide acceptance of MPI. MPI offers communication with different kinds of semantics: point-to-point, collective and one-sided. A communication end-point (usually a process) is referred to using a rank in MPI. Point-to-point operations (Send/Recv) are used to move data between two ranks while collective operations are used to exchange data among a group of processes. In these operations, each rank provides the information about the local source/destination buffers. Point-to-point and collective communication are very commonly used in the HPC applications while the use of one-sided communication has been very limited.

One-sided Communication Semantics in MPI: The MPI one-sided interface enables direct access to the memory of other processes through a “window”. In MPI-2, a window is a region in memory that each process exposes to other processes through a collective operation: MPI_Win_create. MPI-2 one-sided communication interface defines three
types of data transfer operations. MPI_Put and MPI_Get transfer the data to and from a target window respectively. MPI_Accumulate combines the data movement to target with a reduce operation. All of these operations are non-blocking and are not guaranteed to complete, either locally or remotely, until a consequent synchronization operation. The period between two synchronization steps is termed as an epoch. Synchronization modes provided by MPI-2 can be classified into passive (no explicit participation from the target) and active (involves both origin and target). In the passive mode, an epoch is bounded by calls to MPI_Win_lock and MPI_Win_unlock and as the name suggests, this does not require any participation from the remote process. In MPI-3, the one-sided standard has been extended mainly with the addition of new window creation routines, synchronization operations, request-based operations and atomics. Two new window creation mechanisms have been introduced: MPI_Win_allocate allocates the memory required for the window unlike MPI_Win_create which required the user to pass allocated memory; MPI_Win_create_dynamic allows users to dynamically attach and detach memory from a window through MPI_Win_attach and MPI_Win_detach calls. MPI_Rput, MPI_Rget, MPI_Raccumulate, MPI_Rget_accumulate behave like the corresponding regular communication operations but return request objects which enable the user to check for individual local completions. MPI_Win_flush_local and MPI_Win_flush calls have been introduced, which enable checking of bulk local and remote completion of communication operations to a target on a particular window without ending the access epoch. These calls can be used with passive synchronization only. MPI_Win_flush_all and MPI_Win_flush_local_all provide similar functionality but for a window at all the target processes. The complete list of synchronization calls in MPI-3 one-sided model are shown in Figure 2.1(a).
MPI-3 introduced a unified memory model to enable efficient implementations on cache-coherent systems. In this model, the target is guaranteed to eventually get updated data from memory after the operation initiated by an origin process has completed, without requiring any other MPI calls. It also enables ordering of overlapping accumulate operations in an epoch by default. All other operations are unordered. The user can relax the ordering by specifying appropriate info arguments during the window creation process. The new standard deems overlapping accesses as undefined rather than erroneous. This resolves compatibility issues with other programming models/languages which control the semantics of such accesses at a higher level (compiler/language).

**Overlapping Computation and Communication with MPI:** Improving the communication and computation overlap is an important goal in parallel applications and there are different methods to achieve this. Many applications are written with MPI blocking send-receive semantics due to its simplicity. However, blocking primitives require that communication must complete before the process can proceed, so it cannot provide any overlap. Traditionally, MPI non-blocking primitives have been used to alleviate this. Particularly, there is a good chance to have the computation and communication overlap by issuing multiple non-blocking send-receive calls followed by computation on independent data. Furthermore, if receive calls are posted before the corresponding send calls are posted, more overlap can be obtained. This is an active area of research, and several researchers have explored application-level re-design and impact of overlap coupled with re-designed protocols within the MPI library along with improved designs of networking architectures [22, 47, 61, 76, 88, 90].
The one-sided model decouples communication and synchronization, so it is non-blocking in nature and we can utilize it to achieve communication and computation overlap. For example, if the target can post MPI\_Win\_post beforehand, one-sided operations issued by the origin can proceed simultaneously with the following computation. This scenario is shown in Figure 2.1. Of course, this requires the support from network hardware which can move the data without interrupting the host processors using Remote Direct Memory Access (RDMA). All our target platforms have InfiniBand which provides RDMA features. In recent years, there has been some work by application scientists to exploit the relatively newer MPI-2 semantics. Mirin, et al. [59] explore the use of MPI one-sided semantics coupled with multi-threading to optimize the Community Atmosphere Model. Hermmans, et al. investigate the performance of one-sided communication alternatives in the NAS Parallel Benchmark BT application running on 256 cores of a Blue Gene/P in [37].

We discuss the use of MPI one-sided synchronization operations in the context of nearest-neighbor communication, which is a common pattern in scientific applications. This is the case with AWP-ODC, the application we consider in our study. The complete list of synchronization options available in MPI-3 is shown in Figure 2.1(a). In the scenario of near-neighbor communication, MPI\_Win\_fence will result in unnecessary synchronization of all processes when only localized synchronization is required by the application. The Post-Wait/Start-Complete semantics suit this application because each process can create a group including only its neighbors and post/access windows based on this group. We note that passive lock-unlock provides point-to-point semantics, and can potentially be used in this application scenario. In our work with AWP-ODC, we have chosen to use Post-Wait/Start-Complete semantics. Our justification is presented in Section 6.1.4.
Figure 2.1: MPI-2 One-Sided Model and Communication using Post-Wait/Start-Complete Semantics

**MVAPICH2 MPI Library:** MVAPICH2 [9, 69], is an open-source implementation of the MPI-3 specification over modern high-speed networks such as InfiniBand, 10GigE/iWARP and RDMA over Converged Ethernet (RoCE). MVAPICH2 delivers best performance, scalability and fault tolerance for high-end computing systems and servers using InfiniBand, 10GigE/iWARP and RoCE networking technologies. This software is being used by more than 2,100 organizations world-wide in 72 countries and is powering some of the top supercomputing centers in the world, including the 7th ranked Stampede, 11th ranked Tsubame 2.5 and 16th ranked Pleiades.
MPI libraries typically use the *eager* protocol for small messages and the *rendezvous* protocol for large message communication operations. MVAPICH2 uses an RDMA-based eager protocol called RDMA-Fast-Path, along with various optimizations to improve the latency of small message point-to-point communication operations. For large messages, MVAPICH2 uses zero-copy designs based on RDMA-Write or RDMA-Read operations to achieve excellent communication bandwidth. Further, MVAPICH2 offers good scalability through advanced designs such as eXtended RC (XRC), Shared-Receive Queues (SRQ) and Hybrid (UD/RC) communication modes. MVAPICH2 also provides optimized collective communication using shared memory based designs. It also employs different collective algorithms based on the message and job sizes. MVAPICH2 implements one of most optimized implementations of MPI one-sided communication available on InfiniBand. Many implementations of one-sided communication are built on top of two-sided primitives. MVAPICH2 directly takes advantage of RDMA to implement Put and Get operations there by enabling applications to achieve overlap using one-sided semantics. However, direct RDMA based implementation is limited to active and flush synchronization at the time this report is written [38, 77].

### 2.2.2 PGAS Models and OpenSHMEM

Partitioned Global Address Space (PGAS) models provide a shared memory-like abstraction that makes it easier to program applications with irregular communication patterns. They also ensure performance by exposing key locality information to the application developer. This has made them an attractive alternative to traditional message passing models like MPI. SHMEM has been a popular PGAS model. There have been several successful implementations of SHMEM including Cray SHMEM, SGI SHMEM, Quadrics.
SHMEM and others [81]. However, each of these implementations have their own variation of semantics making it hard for SHMEM programs to be portable. OpenSHMEM [66] is an effort to bring together these different implementations under an open standard and make SHMEM more widely useful for the community. The OpenSHMEM programming model operates on a symmetric memory address space. It allows processes or processing elements (PE) to see each other’s variables with a common name, each PE having its own local copy of the variables. These are called symmetric variables and are allocated collectively by all the PEs at the same point of the execution path. In C, symmetric objects can be global or static variables and can also be allocated dynamically from a symmetric heap using routines like shmalloc and shmemalign. OpenSHMEM defines point-to-point (contiguous and strided put/get) and collective communication operations for data movement between symmetric variables. The put operations in OpenSHMEM return when the data has been copied out of the source buffer; they need not be complete at the target. The completion at target is ensured using explicit point-to-point and collective synchronization routines. The get operations return only when data is available for use in the local buffer and hence do not require additional synchronization. OpenSHMEM also provides atomics and lock routines that allow implementation of critical regions.

2.3 Communication on Clusters with GPUs

Clusters with GPUs and InfiniBand interconnect are becoming increasingly common. While GPUs boost the compute power of the nodes, IB provides low latency and high bandwidth communication between them. NVIDIA has been the front runner in GPU technology and its GPU devices attributed to a large portion of GPUs used in HPC clusters. CUDA and OpenCL have been the most widely used programming frameworks for GPUs.
are connected as peripheral devices on the I/O bus (PCI express), similar to IB adapters. To transfer data between GPUs on different nodes on current GPU clusters, data has to be first copied onto the local host using CUDA or OpenCL, transferred over IB to the remote host and then copied onto the remote GPU using CUDA or OpenCL.

CUDA provides a feature called Unified Virtual Addressing (UVA). UVA provides a single address space for all CPU and GPU memory that is allocated by a process. In addition, UVA can also be used to find out if a particular buffer was allocated in the GPU memory or in the host memory. This is very useful for communication libraries to detect the location of a communication buffer and handle data movement accordingly. Like IB, CUDA also requires host buffers to be registered in order to support efficient data movement between GPU and the host. In early versions of CUDA, both CUDA and IB could not register the same region in host memory. This required an additional copy on the host between the copy from the GPU to the host and the network transfer. This severely limited the performance of data movement between GPUs in a cluster. This has improved with the introduction of GPUDirect Technology.

**NVIDIA GPUDirect:** NVIDIA’s GPUDirect technology provides a set of features that enabled efficient communication among GPUs and between GPUs and other devices. The initial version of GPUDirect, release with CUDA 4.0, enabled the same host memory regions to be registered by both an InfiniBand adapter and a GPU device. This avoids the additional copy in host memory.

**CUDA IPC:** In CUDA 4.1, GPUDirect had been extended with a new interface (cuIpc*/cudaIpc*) to support efficient inter-process communication on GPU device memory. This enables a process to directly expose its GPU device buffer to a remote process. The remote process can map this buffer into its address space and can issue operations like
cuMemcpy directly over it. To achieve this, a process creates an Inter-process Communication (IPC) handle on its device buffer by using cuIpcGetMemHandle. It then sends this handle to the remote process through host-based communication. The remote process calls cuIpcOpenMemHandle on this handle to map the associated buffer into its local address space. The copies involving IPC buffers can be asynchronous in nature even when synchronous calls like cudaMemcpy are used. So a host-based synchronization between processes may not ensure the completion of preceding IPC operations.

The events interface in CUDA has been extended to ensure ordering of IPC operations with other CUDA calls. A process can create an event for IPC communication by using cuEventCreate with CU_EVENTInterstitial flag. It can then create an IPC handle on this event with a call to cuIpcGetEventHandle. This handle can be mapped by a remote process using cuIpcOpenEventHandle. Now, both source and remote processes can use this event to order their operations with respect to IPC operations between them. A process can record an event on a stream using cuEventRecord. In CUDA, a stream is a sequence of commands that execute in order. The event is recorded on the device only after all the preceding operations on the stream have completed. The remote process can issue a cuStreamWaitEvent to force any future operations to happen only after the event has been recorded on the device. The ordering is enforced on all the streams when the stream specified is NULL. The cuStreamWaitEvent call orders events on the device and does not block the CPU until the event happens. This minimizes the overhead observed by the CPU.

**GPUDirect RDMA:** In CUDA 5.0, GPUDirect has been extended to allow third party PCIe devices, including InfiniBand adapters, to directly read/write data from/to GPU device memory. This is called GPUDirect RDMA (GDR). GDR allows data from the GPU to be
directly sent over the network without being staged through host memory. Though GDR provides a low latency path for inter-node GPU-GPU data movement, its performance for large data transfers is limited by the bandwidth supported for PCIe peer-to-peer exchanges on modern node architectures [72]. The performance of peer-to-peer transfers in MB/sec and as a percentage of peak FDR IB bandwidth, is presented in Table 2.1. We consider scenarios where the IB card and GPU are connected on the same socket or on different sockets. These limitations severely limit the performance achieved by GPUDirect RDMA, for large message transfers. On Intel SandyBridge (SNB), performance of peer-to-peer reads is severely limited, with a peak bandwidth of <1GB/s. This is a mere 13% of the peak performance that can be achieved on a PCIe bus or with an FDR InfiniBand adapter. This severely limits the performance when data from the GPU has to be transferred onto the network. Peer-to-peer write performance is only partially limited, with peaks reaching up to 83% of the optimal channel bandwidth. On Intel IvyBridge (IVB), these bottlenecks have been alleviated to some extent but peer-to-peer read bandwidth is still limited to only 50% of the channel’s peak bandwidth. The performance discussed here is when GPU and IB card are connected to the same socket. Performance of both peer-to-peer write and read operations is severely limited when the devices are connected to different sockets. It is to be noted that these artifacts are specific to the node architecture and not the GPU or the IB adapter. Similar limitation can arise between any two PCIe devices which involve peer-to-peer transfers.

OpenCL does not provide advanced GPUDirect techniques like IPC and GDR yet. So, movement of data in a GPU cluster has to be staged through the host. It also does not provide UVA. Hence, communication libraries have to use other techniques to differentiate between CPU and GPU buffers.
### Table 2.1: Peer-to-Peer Performance on Intel Architectures and Percentage of Bandwidth offered by FDR IB Adapter (6397 MB/s)

<table>
<thead>
<tr>
<th>P2P Read</th>
<th>SandyBridge</th>
<th>IvyBridge</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intra-Socket</td>
<td>Inter-Socket</td>
</tr>
<tr>
<td>P2P Read</td>
<td>962 MB/s (15%)</td>
<td>370 MB/s (6%)</td>
</tr>
<tr>
<td>P2P Write</td>
<td>5280 MB/s (83%)</td>
<td>1075 MB/s (17%)</td>
</tr>
</tbody>
</table>

**MPI and OpenSHMEM Models on GPU Clusters:** Though CUDA or OpenCL is used to express parallelism on GPUs, developers have to depend on higher level programming models like MPI or OpenSHMEM to express parallelism across a cluster, using processes. However, MPI/OpenSHMEM models and runtimes preceding this work could not be used to moved data that is in GPU memory. For example, MPI Send/Recv calls could only be issued of host buffers while there is no way in OpenSHMEM to allocate communication buffers on device memory. Therefore, CUDA or OpenCL calls have to be used to move data on the host before MPI or OpenSHMEM if used for data movement between processes. This requirement of using two models for moving data makes it harder for communication to be optimized in applications.

### 2.4 Communication on Clusters with Intel MIC based Coprocessors

Xeon Phi is the first co-processor based on the Intel Many Integrated Core architecture. Current generation Xeon Phi (SE10P) coprocessors are co-located on systems as PCI Express (PCIe) devices and are equipped with 61 processor cores that are interconnected by a high performance bi-directional ring. Each core is an in-order, dual-issue core which supports fetch and decode instructions from four hardware threads. With eight memory controllers and 16 memory channels in aggregate, a theoretical bandwidth of up to 352
GB/s is predicted. MIC architecture provides x86 compatibility and supports most of the programming models that are available on hosts, including MPI.

**Compute Modes:** It offers the following modes of operation for the MPI programming model [17]: 1) Offload mode, 2) Coprocessor-only mode, and 3) Symmetric mode. In Offload mode, MPI processes are executed on one of the architectures — either the coprocessors or the host processors. The other architecture is used solely as an accelerator for MPI processes to offload computation onto. This is akin to the usage seen with most GPGPU clusters. In offload mode, all the MPI processes execute either on the host or on the MIC with the other being used as an accelerator. In the Symmetric mode of operation, MPI processes uniformly span the domains of both the host and the coprocessor architectures and hence are amenable to immediate porting of existing MPI applications with little or no modifications. However, applications have to deal with the asymmetry in compute power and memory available on the MIC compared to the host. In the symmetric mode, the MPI job needs to be run in the Multiple Program Multiple Data (MPMD) mode. This is because all binaries which run on the MIC must be cross-compiled on the host for compatibility with MIC architecture. The coprocessor-only mode is a subset of symmetric mode with all MPI processes being confined to the MIC architecture alone.

**Communication Channels:** To support the full spectrum of usage models for MIC, Intel’s Manycore Platform Software (MPSS) offers three modes of inter-process communication: shared-memory channel, Symmetric Communication Interface (SCIF) and IB-verbs based communication. For communicating processes that reside on the same MIC device, POSIX shared memory is supported, along with a multi-threaded memcpy. Alternatively, for communication operations within the same MIC, or between the MIC and host processes, within the same compute node, SCIF generic communication API may be used.
SCIF is a sockets-like API for communication between processes on MIC and host within the same system. SCIF API provides both send-receive semantics, as well as Remote Memory Access (RMA) semantics. Send-receive semantics involve both the processes, which serve as either source or destination, in the communication operation. RMA semantics define operations to register a region of memory in the user space as a window exposed for remote access. Upon registration, further data transfers can take place in a one-sided manner with just one process either reading from a window or writing into one [17].

MPSS provides two ways of using IB verbs for communication on MIC clusters, as depicted in Figure 2.2. This allows applications to natively use MPI implementations that are built on top of InfiniBand verbs API. A direct OFED communication stack is provided to support a symmetric mode of communication on just the MIC or between the MIC and
the host processor. This harnesses the advantages of the physical InfiniBand Host Channel Adapter (HCA) for intra-node and inter-node communication between a MIC and the host or between two MICs. To use IB directly and to enable processes on the MIC to talk with the HCA, Intel has facilitated a proxy based approach where all privileged operations are staged through an IB proxy client, the coprocessor Communication Link (CCL) driver, residing on the MIC to make requests on behalf of the process to IB proxy server running on the host. On completion of these privileged operations ensuing data placement calls from the process on the MIC can be made in a direct manner to the HCA using PCIe peer-to-peer copies. Alternatively, MPSS’s implementation of IB verbs over SCIF API called IB-SCIF may be used. This allows processes to use verbs API over a virtual HCA as all underlying operations are handled using SCIF. This is especially beneficial for MPI processes that reside in the Xeon Phi alone and provides ease of porting existing MPI applications to the MIC in coprocessor only mode.
In this section, we present our idea of using MPI as a unified interface for communication on GPU Clusters. We also highlight the importance of a high-performance MPI runtime and present designs to optimize communication from GPU memory, in different system configurations. We take advantage of the latest techniques offered by the underlying software and hardware on NVIDIA GPUs. Through MVAPICH2-GPU, we provide a high performance and high productivity approach for developers to design communication in MPI applications on GPU clusters. Experimental evaluation shows that handling data movement from GPU memory directly in the MPI library provides significant improvements in communication performance while hiding the complexity from the application developer.

### 3.1 Unified Interface for Data Movement

Figure 3.1 illustrates the data-flow from a GPU device on one node to a GPU device on a remote node, in applications when MVAPICH2-GPU was not available. Firstly, the source process copies data from its GPU device memory to the main memory (1). Then, it sends the data to the remote process (2). Finally, the target process copies the data from its main memory to its GPU device memory (3). The application programmer has to independently
handle the copies from GPU memory to main memory using CUDA and the inter-process transfers through a communication model like MPI. Internally, most MPI implementations can use shared memory or network to transfer data between processes depending on their location.

With MVAPICH-GPU, we have enabled the use of standard MPI interfaces to handle communication involving GPU device memory. In order to directly use GPU memory address as parameters in standard MPI interfaces, we use Unified Virtual Addressing (UVA) provided by CUDA. UVA allows the library runtime to detect if a given memory address is in the host memory or in the device memory. The data movement can be optimized by the MPI library, under the hood, using different techniques. For example, the three data transfers: GPU memory to host memory, network, host memory to GPU memory (on remote side) are all pipelined. This pipelining is internal to MVAPICH2-GPU. The application programmer remains unaware of this complexity. For the programmer, it is simply a normal MPI communication call.

Figures 3.2 and 3.3 illustrate pseudo code that compare the existing methods of GPU-to-GPU communication with that offered by MVAPICH2-GPU. Figure 3.2(a) shows a
naive implementation using blocking CUDA memory copies and blocking MPI communication calls. Figure 3.2(b) shows how an application developer can improve the performance by carefully interleaving non-blocking memory copy and communication calls. Application and platform specific tuning is crucial to achieve maximum performance in this scenario. Figure 3.3 shows the ease in which a user can exploit overlap using MVAPICH2-GPU: the standard MPI_Send and MPI_Recv interfaces are used; and the underlying library takes care of optimally pipelining the memory copies and network transfers. MVAPICH2-GPU not only simplifies programming, but also has the potential for better performance as discussed in the following sections.

3.2 Inter-node Communication

In this section, we discuss designs for efficient inter-node communication between GPUs on a cluster. We first present a design that pipelines CUDA memory copies and network transfers to optimized communication. Then, we present a design that takes advantage of the new GPUDirect RDMA technology introduced with CUDA 5.0.

3.2.1 Pipelined Communication

We describe the pipeline design in MVAPICH2-GPU library for moving data from one GPU device’s memory to remote GPU device’s memory. The steps are illustrated in Figure 3.4. The sender sends a Request To Send (RTS) message to initialize the data transfer. After the receiver receives the RTS, it replies back with a Clear to Send (CTS) (based on MPI message matching semantics). The receiver also encodes the remote buffer address and size of the message to be received within the CTS message. The data is internally buffered at the sender and receiver host memory. These buffers are called vbufs, and are managed in a FIFO pool. The address encoded in the CTS message is that of a list of vbufs.
if(rank == 0) {
    cudaMemcpy(s_buf, s_device, size, cudaMemcpyDeviceToHost);
    MPI_Send(s_buf, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
    ...
} else if(rank == 1) {
    MPI_Recv(r_buf, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD, &reqstat);
    cudaMemcpy(r_device, r_buf, size, cudaMemcpyHostToDevice);
    ...
}

(a) Naïve MPI and CUDA without pipelining (Good Productivity, Bad Performance)

if(rank == 0) {
    for(j=0; j<pipeline_length; j++) {
        cudaMemcpyAsync(s_buf + j*block_size, s_device + j*block_size,
                        block_size, cudaMemcpyDeviceToHost, cuda_stream[j]);
    }
    for(j=0; j<pipeline_length; j++) {
        while (result != cudaSuccess) {
            result = cudaStreamQuery(cuda_stream[j]);
            if(j>0) MPI_Test(sreq[j-1], &flag, &status[j-1]);
        }
        MPI_Isend(s_buf + j*block_size, block_size, MPI_CHAR, 1, 1,
                   MPI_COMM_WORLD, &sreq[j]);
    }
    MPI_Waitall(pipeline_length, sreq, sstat);
    ...
} else if (rank == 1) {
    for(j=0; j<pipeline_length; j++) {
        MPI_Irecv(r_buf + j*block_size, block_size, MPI_CHAR, 1, 1,
                   MPI_COMM_WORLD, &rreq[j]);
    }
    for(j=0; j<pipeline_length; j++) {
        MPI_Wait(&rreq[j], &rstat[j]);
        cudaMemcpyAsync(r_device + j*block_size, r_buf + j*block_size,
                         block_size, cudaMemcpyHostToDevice, cuda_stream[j]);
    }
    for(j=0; j<pipeline_length; j++) {
        cudaMemcpyAsync(r_device + j*block_size, r_buf + j*block_size,
                         block_size, cudaMemcpyHostToDevice, cuda_stream[j]);
    }
    cudaMemcpyAsync(r_device + j*block_size, r_buf + j*block_size,
                         block_size, cudaMemcpyHostToDevice, cuda_stream[j]);
    cudaStreamSynchronize(cuda_stream[j]);
    ...
}

(b) Advanced MPI and CUDA with pipelining (Low Productivity, Good Performance)

Figure 3.2: Pseudocode for existing approaches for data movement on GPU clusters
When the CTS message is received by the sender, it starts asynchronous CUDA memory copy from device to host for each block (pipeline unit). The MPI library uses CUDA stream query function to check the status of each asynchronous memory copy. When one of the asynchronous memory copies finishes, the sender calls InfiniBand verbs interface to perform the RDMA write. Finally, after each RDMA write finishes, the sender sends out a RDMA finish message. When the receiver gets the RDMA finish message, it starts the asynchronous CUDA memory copy to move data from a vbuf to GPU device memory.

The above description is for the case where GPU direct is enabled. If GPU-Direct is not enabled, we use an additional shadow vbuf to do one more memory copy. This additional memory copy is required since buffers cannot be shared between InfiniBand and GPU devices. However, this is not required on most clusters as GPUDirect has become commonly available.

In the internal pipelined design, consideration of two factors is important for performance: the first factor is vbuf organization and its usage. With traditional MPI and CUDA interfaces, some portion in main memory is allocated to communicate with GPU device’s memory. The allocation, management and use of this memory is directly in the programmer’s control. In our design, MVAPICH2-GPU is responsible for the allocation and management of this memory. The vbuf is allocated and freed in MPI_Init() and MPI_Finalize(), respectively. Once vbuf is allocated, it will be divided into blocks and organized as a buffer pool. During the handling of RTS, one vbuf block will be taken from the buffer pool. In order to keep total memory usage in control, one buffer pool is maintained at each process ($O(N)$) rather than keeping one buffer pool for each peer process involved in communication ($O(N^2)$). The size of each buffer block may be smaller...
than the size of data to be sent as vbuf block. Additionally, vbufs are managed as a FIFO pool for RDMA.

The second factor is the usage of CUDA API for asynchronous memory copies. We use `cudaStreamQuery()` to do busy checking instead of blocking `cudaStreamSynchronize()` call. The CUDA stream query interface is a non-blocking operation to check whether pending DMA transactions are over. It returns quickly, regardless of whether a copy is complete or not. In contrast, the CUDA stream synchronize call blocks for particular copy to finish. In our design, the control returns back to MPI progress engine to handle other messages, which essentially avoids blocking of pipeline.

```c
if(rank == 0) {
    MPI_Send(s_device, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
    ...
} else if(rank == 1) {
    MPI_Recv(r_device, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD, &reqstat);
    ...
}
```

Figure 3.3: Pseudocode using MVAPICH2-GPU (Good Performance and Good Productivity)

**Choosing Optimal Pipeline Block Size:** In the naive approach, where pipelining is not employed, the latency to transfer data of size N, from GPU device memory to the remote GPU device memory, is: \( T_{cuda}(N) + T_{mpi}(N) + T_{cuda}(N) \). \( T_{cuda} \) represents time spent in CUDA memory copies and \( T_{mpi} \) represents time spent in MPI library for the network transfer. Pipelining achieves best performance when latency of network transfer and latency of CUDA memory copies are overlapped by one another. The expression to
Figure 3.4: MVAPICH2-GPU with GPU-Direct
model the latency of pipelined data transfer, where the data is divided into $n$ blocks is:

$$ T_{\text{cuda}}(N/n) + n \times T_{\text{mpi}}(N/n) + T_{\text{cuda}}(N/n) $$

Figure 3.5 illustrates the latency for network transfer and the CUDA memory copy on the cluster with CUDA 4.0 and GPUDirect. We observe that, when the data size is equal to 128KB, the network (RDMA write) latency is equal to CUDA memory copy latency on this platform. It can be expected that MVAPICH2-GPU will get better performance when the data size is larger than 128KB where CUDA memory copy latency can be overlapped by RDMA write latency. Based on our experimentation, we found 256KB to be optimal for TACC Longhorn, the cluster used to study these designs. The pipeline unit is presented as a configurable parameter of the MVAPICH2 library. This can be tuned once by the system administrator during the time of installation of the library by using OSU micro-benchmarks. Once the optimal value for the cluster is found, it can be placed in a configuration file (MVAPICH2 supports this), and end-users will transparently use this.

![Figure 3.5: Comparison and Cross-over between IB RDMA Transfer and CUDA Memory Copy](image)

(a) Small Message Latency  
(b) Large Message Latency
3.2.2 Designs using GPUDirect RDMA

GPUDirect RDMA enables InfiniBand adapters to directly read from and write into GPU device memory. MPI libraries typically use different designs for communication, depending on the size of the message being transferred. We analyze how these design choices apply when GDR and GPU device buffers are involved. For small messages between host buffers, the overhead of synchronization between processes can incur a considerable overhead given the actual data movement involved is small. So, avoiding any synchronization overheads is one of the key considerations for small message communication designs. In IB Send/Recv semantics, sender and receiver processes should post matching send and receive requests with the buffer information to transfer data directly between the source and destination buffers. The receive should be posted before the send operation. In IB RDMA semantics, where one process provides both the source and target buffer, the buffer address and registration information has to be exchanged between the processes before data can be initiated. To avoid these synchronization overheads, MPI libraries use the eager protocol where data is staged through pre-determined intermediate buffers at the receiving process. The data is copied over to the ultimate destination buffer from the intermediate buffers when the receiving process is free to do so.

Most MPI libraries use RDMA to implement small data transfers as it does not require the receiver process to post and handle any IB requests. Instead, the sender process sets a flag as part of the data written to the staging buffer. Since the usage of buffers is pre-arranged, the receiver process polls on the corresponding memory location to detect the arrival of a new message. Both memory polling and memory copy do not pose significant overheads in the case of host buffers. However, these considerations change when GPU buffers are involved. A CPU process polling on GPU memory is prohibitively expensive.
Also, an extra copy from either a staging Host or GPU buffer to the destination GPU buffer nullifies or significantly reduces the benefit of using GDR. The polling overhead can be avoided by implementing the eager protocol using IB Send/Recv operations instead of RDMA and pay for the low request handling overhead. However, the extra copy will still be required. Another alternative we have considered is use of the RDMA Immediate operation provided by IB which is an RDMA operation that generates a completion at the receiver. This removes the need for memory polling but will still require the additional copy if a staging buffer is used.

Owing to these limitations, we use the rendezvous protocol to implement all message transfers using GDR. The sender and receiver process exchange control messages containing the address and registration information of the source/destination buffers before the actual data is transferred using RDMA operations. The path for data movement using GDR that bypasses host memory at the sender and the receiver is depicted in Figure 3.6(a). The control messages are exchanged between the host memories and their overhead is low compared to the data movement from GPU memory. IB registration of GPU memory is expensive as in the case of host memory. Hence, we use registration cache to store registration entries for reused buffers, thus hiding this overhead. MVAPICH2 uses a memory management layer called Ptmalloc [94] to handle any malloc and free calls of host memory that are made in an application. It un-registers any registered memory regions before they are actually freed. Since GPU device memory allocation is handled by the CUDA library, we currently trap the cudaFree calls (or their driver equivalents) to ensure that the memory is unregistered before the memory is freed. Freeing and re-allocation of CUDA device memory that is still registered with the IB adapter leads to undefined behavior in the current
OFED stack. A mechanism that provides a call back to the MPI library or automatically unregister any registered memory when it is freed is desirable for portability.

While using GDR with rendezvous protocol gives a low latency path for small message transfers from GPU memory, it provides limited bandwidth because of the chipset limitations on modern Intel architectures as explained in Section 2.3.

**GPUDirect RDMA Hybrid:** To overcome the bandwidth limitations posed when using GDR on modern Intel architectures, we propose a hybrid solution between the existing host-based pipeline solution in MVAPICH2 and the new GDR based design. For small messages, where latency is critical, we use the fast GDR path and for large messages we switch back to the existing host-based solution which circumvents the P2P read limitation. The threshold for the switch between these two solutions depends on the performance characteristics of an underlying platform.

**GPUDirect RDMA Hybrid Advanced:** As outlined in Section 2.3, the bandwidth limitation in GDR based internode GPU-to-GPU communication appears to arise from the limited P2P read bandwidth offered by the underlying chipset. This affects the part of the transfer where the network adapter reads data from GPU memory before sending it over the network. The bottleneck when the adapter writes to GPU memory is not severe from the performance results of host-to-GPU transfers. Based on these observations, we propose an improved pipeline design in MVAPICH2 in which, the data is copied onto the host at the sender and then is directly written to the destination GPU memory at the receiver. These steps are done in a pipelined fashion as shown in Figure 3.6(b). Using GDR, we bypass the host on the receiver process, thus reducing inefficiencies. By using host-based staging, we avoid the bottleneck in P2P reads at the sender process.
Figure 3.6: Proposed GDR based Inter-node GPU-to-GPU Pipelined Data Movement in MVAPICH2
3.3 Intra-node Communication on Multi-GPU Clusters

In this section, we discuss designs for efficient intra-node GPU-GPU communication when there are multiple GPUs per node. One design alternative is to pipeline CUDA memory copies between GPU and host with shared memory copies between processes, similar to the design discussed for inter-node communication. The approach discussed here takes advantage of the CUDA Inter-Process Communication (IPC) feature introduced in CUDA 4.1. IPC enables direct data movement between GPUs connected to the same socket/IOH completely by-passing the host.

3.3.1 Designs using CUDA IPC

Most of the state-of-the-art MPI implementations for multi-core systems employ different schemes for intra-node two-sided communication depending on the message size. Eager protocol is used for short messages ($\leq 17$KB in MVAPICH2, for the platform used) and Rendezvous protocol is used for large messages ($>17$KB in MVAPICH2, for the platform used). This is because synchronization overheads between sender and receiver have a considerable impact on latencies for small messages while large message transfer latencies are dominated by the data copy cost. This behavior applies to GPU-to-GPU communication as well. The asynchronous nature of CUDA IPC copies requires additional synchronization to be added into the designs for GPU-to-GPU communication.

Short Message Communication: The current implementation of intra-node MPI communication between GPUs in MVAPICH2 is done by staging data through buffers in main memory. The source process copies data out from its device memory into the main memory buffer that it shares with destination process (eager host buffers). The destination process copies the data from the shared memory buffer to its device memory. We outline the new
design using CUDA IPC below. To eliminate synchronization overheads for short messages, we use pre-allocated GPU device buffers (eager device buffers) to stage GPU-to-GPU communication between processes. CUDA Inter-process communication handles on these buffers are created, exchanged and mapped during the MPI initialization phase. These eager device buffers are designed as extensions to host buffers that are used for eager messaging. This addresses two concerns. First, it avoids polling directly on the device memory which can be detrimental to performance. Secondly, it has only one point of polling for both host-host transfers and transfers involving GPUs. This avoids the need for explicit mechanisms like sequence numbers to satisfy MPI message ordering.

When a source process A has a message to send from its GPU device memory to a destination process B, it copies the data into Process B’s eager device buffers. It then prepares a header with a flag set to indicate that the data is on the device and copies it into the process B’s eager host buffers. When Process B has a matching receive and detects the header from process A, it copies the data from its device eager buffers into the destination buffer. The device eager buffers are allocated in a pair-wise fashion and are managed as circular buffers with head and tail pointers to co-ordinate their use. As the number of buffer pools is limited to the number of devices/processes per node, allocating pair-wise buffer pools does not pose a major scalability bottleneck.

IPC communication in CUDA can be asynchronous with respect to the CPU even when the synchronous CUDA memory copies are used. Hence the transfer and detection of headers in the eager host buffers can overtake the IPC copy to the remote eager device buffers. We avoid this by using a pre-allocated pool of CUDA inter-process events (IPC Events) between process pairs. After process A copies the data into Process B’s device buffer, it records an event by calling cuEventRecord. Process B issues a
`cuStreamWaitEvent` on the same event when it detects the header and issues a copy from the eager device buffer to the destination buffer. This `cuStreamWaitEvent` call orders the two copies on the device.

A similar synchronization is required when a process A wants to reuse an eager device buffer at Process B. This is done by using record and wait calls in the opposite direction: Process B records an event whenever it updates the tail and Process A waits on this event whenever it updates its count of available buffers. CUDA runtime automatically ensures ordering between operations issued by the same process. Explicit synchronization is necessary only with respect to IPC copies. The mechanism of the design presented above is depicted in Figure 3.7.

Figure 3.7: Steps involved in an intra-node eager GPU-to-GPU communication from Process A to Process B: (1) Process A does IPC copy from source device buffer to eager device buffers at Process B, (2) Process A calls `cuEventRecord` the IPC event, (3) Process A copies the header into eager host buffers at Process B, (4) Process B detects the header and calls `cuStreamWaitEvent` on the IPC event, and (5) Process B issues a copy from its eager device buffers to the destination eager buffer.
Large Message Communication: MVAPICH2 currently uses copies through the host memory for large message GPU-to-GPU communication as well. However, unlike the eager protocol, there is a control message handshake (Request To Send (RTS) and Clear To Send (CTS)) between the communicating processes before the actual copies happen. This is to avoid flooding of the shared memory buffers with data for which receives have not been posted. In the modified design, source process creates an IPC memory handle on the source device buffer and sends it along with the RTS message. When a matching receive is posted on the destination process, the destination process maps the handle it received and copies data directly from source process’s device memory to its own device memory using cudaMemcpy. After the copy is issued, the destination process records an IPC event and sends back a FIN message to the source. The sender then issues a cuStreamWaitEvent call on the event to ensure that the IPC copy completes before any following accesses to the device buffer are executed.

The cost of creating and mapping memory handles poses a significant overhead and can negate the benefits gained by the reduced number of copies in IPC communication. We present an analysis of this overhead in Section 3.4.2. To avoid this cost, we maintain a handle cache (a list of key, value pairs) which keeps track of IPC handles created locally and handles that were mapped from each remote process. CUDA IPC memory handles can only be created on the base address of a device memory allocation. When an MPI_Send call is issued at the source process, MVAPICH2 retrieves the base address of the corresponding memory region allocation using cuMemGetAddressRange. It searches the cache to see if a handle had been created on this region. When a match is not found, it creates a new handle, stores it in the cache and sends it to the destination process. The destination process performs a similar lookup by handle before mapping a new handle and caches
any newly mapped handles and corresponding addresses. A wrapper around `cuMemFree` call is used to detect freeing of any memory regions. The memory handles are closed and corresponding cache entries are evicted at both local and remote processes before the buffer is actually freed.

### 3.4 Experimental Evaluation

We evaluate the proposed designs for internode and intranode communication on various clusters using different benchmarks, application kernels and applications. We present details of the experimental setup and performance results in the following sub-sections.

#### 3.4.1 Internode Communication

**Evaluating Pipelining Designs:** The cluster used in our experiments has eight nodes equipped with dual Intel Xeon Quad-core Westmere CPUs operating at 2.53 GHz and 12 GB of host memory. These nodes have Tesla C2050 GPUs with 3 GB DRAM. The InfiniBand HCAs used on this cluster are Mellanox QDR MT26428. Each node has Red Hat Linux 5.4, OFED 1.5.1, MVAPICH2-1.6RC2, and CUDA Toolkit 4.0. All the experiments were run with GPUDirect. Our MPI-level evaluation is based on OSU Micro Benchmarks [67]. We run one process per node and use one GPU per process for all experiments. We run the collective tests on eight nodes.

**Point-to-point Performance using Pipelining Designs:** We compare the performance of data movement using existing approach of using CUDA and MPI with data movement using MVAPICH2-GPU. `Memcpy+Send` is the method illustrated in Figure 3.2(a). It uses blocking calls: `cudaMemcpy`, `MPI_Send`, and `MPI_Recv`. `MemcpyAsync+Isend` is the method shown in Figure 3.2(b). It uses the corresponding non-blocking calls to implement the pipeline and achieve overlap. Finally, MVAPICH2-GPU represents the method using
the integrated MPI and CUDA design proposed in this chapter where MPI is used for communication directly from GPU device memory. The results presented in Figure 3.8 compare GPU to GPU send/recv latencies using the different approaches. The results show that MVAPICH2-GPU consistently performs better than the optimized MPI-level implementation (i.e. optimizations are done by the application developer), MemcpyAsync+Isend. We achieve up to 24% improvement in latency, for message size of 4MB. Comparing with Memcpy+Send, the benefit is 45%. MVAPICH2-GPU performs better than the best achievable performance at MPI-level as it avoids the overhead of MPI-level rendezvous handshakes, message matching and message processing for each of the pipeline messages. Moreover, implementing the pipeline internally enables us to issue non-blocking CUDA memory copy and RDMA write in an overlapped and fine-grained fashion. This improves the achievable overlap.

Figure 3.8: Two-sided Latency Performance using Pipelined Designs
**Collective Performance using Pipelining Designs:** Figure 3.9 compares the performance of three commonly used collective operations: Scatter, Gather, and Alltoall. Because of the blocking nature of collectives and their complex data exchange patterns, it is hard to pipeline the communication with the asynchronous CUDA memory copy by just using MPI interface (without CUDA integration). Often, the most optimal collective algorithm is platform dependent, since MPI library designers have spent a lot of effort to optimize it. For example, in Alltoall alone, MVAPICH2 uses three different algorithms based on the message size and the number of processes. In these figures, cudaMemcpy+collective is the method to copy the data in the GPU memory into the main memory, do the collective, and then copy the data back to the GPU memory. We can observe a performance improvement of up to 32% for Scatter, 37% for Gather, 30% for Alltoall at message sizes of 4MB, 1MB and 4MB, respectively.

**Evaluation of Designs using GPUDirect RDMA:** We use a local cluster with two Sandy Bridge and two WestmereEP nodes for our tests. The Sandy Bridge nodes have 16 Xeon E5-2670 2.60GHz cores (two sockets with eight cores each) and 32GB of memory. They support PCIe 3.0 and are equipped with an NVIDIA Tesla K20c (Kepler) and a Mellanox ConnectX-3 IB FDR MT4099 adapter. The WestmereEP nodes have 12 Xeon E5645 2.40GHz cores (two sockets with six cores each) and 24GB of memory. They support PCIe 2.0 and are equipped with the same GPU and IB adapter as the other nodes. All the nodes run RHEL 6.0 and OFED-1.5.4.1 with the GPUDirect RDMA patch. Our designs are based on the MVAPICH2 1.9b release code and we have used OSU Micro-benchmarks (OMB) 3.9 for our experiments. The OMB point-to-point benchmarks provide options to evaluate communication from GPU device memory. It is to be noted that OMB bandwidth benchmarks report performance in Million Bytes/sec and these are represented as ”MB/s”
Figure 3.9: Collectives Latency Performance using Pipelined Designs
in the graphs. We have extended the collective benchmarks to measure performance with GPU device memory buffers. For all our experiments we run one MPI process per GPU or node. The point-to-point experiments were run between two nodes and all the collective and application runs were carried out on four nodes. When presenting the results, we call the existing host-based pipelining designs discussed earlier as 'MV2'. These are available in the MVAPICH2 1.9 release. We call the GPUDirect, GPUDirect Hybrid and GPUDirect Hybrid Advanced designs presented in Section 3.2.2 as ‘MV2-GDR’, ‘MV2-GDR-H’ and ‘MV2-GDR-H-Advanced’, respectively.

**Point-to-point Performance of Designs using GPUDirect RDMA:** Figure 3.10 presents a comparison of inter-node GPU-to-GPU MPI latency for various message sizes, using the different designs. For small messages, we see that all GDR based designs clearly outperform the existing host-based approach. They deliver a 69% improvement in latency for 4Bytes messages. However, for medium and larger messages, where MVAPICH2 uses a pipeline to optimize host-based transfer and where the P2P read bottleneck occurs, we see that ‘MV2-GDR’ performs considerably worse compared to the existing ‘MV2’ design. ‘MV2-GDR-H’, which switches between the above two designs, delivers the best performance of these two designs, for all the message sizes. ‘MV2-GDR-H-Advanced’ outperforms others for medium messages as it always uses GDR at the receiver process, thus reducing any host staging and pipelining overheads. It outperforms the next best design by 32% for 128KB messages. For large messages, where pipelining overheads seem to be become less prominent, the improvements from ‘MV2-GDR-H-Advanced’ are not visible in the graph. ‘MV2-GDR-H-Advanced’ shows a 8% improvement for 2MB messages.

The optimal block size used in the ‘MV2’ design for a latency test differs from one that is optimal for a bandwidth test. For latency tests, we have observed 64KB blocks to be
optimal while for bandwidth tests, 1MB blocks delivered the best performance. We have used these settings for the corresponding experiments. For ‘MV2-GDR-H’, the switch from GDR based design to host-based design also varied between latency and bandwidth tests. This is because of the bandwidth limitation imposed by the underlying architecture. For latency experiments, where bandwidth limitations show up only for very large messages, a 32KB threshold delivered the best performance. For bandwidth tests, the optimal threshold was 8KB. We use the corresponding thresholds in our experiments and we believe a similar variation will occur for applications depending on their nature of communication.

![Figure 3.10](image)

(a) Small Messages  
(b) Medium Messages  
(c) Large messages

Figure 3.10: Comparison of Latency Performance Using the Existing Host-based Pipelining and the Proposed GDR-based Designs on Sandy Bridge
We present a comparison of bandwidth performance in Figure 3.11. For small messages, all the GDR based designs benefit from the host bypass, providing a 2x improvement in bandwidth for 4KB messages. For medium and large messages, we see that ‘MV2-GDR’ is limited by the P2P read bandwidth offered by the underlying platform, as outlined in Section 2.3. ‘MV2’ and ‘MV2-GDR-H’ circumvent this issue by pipelined staging through the host. ‘MV2-GDR-H-Advanced’ benefits from the use of GDR at the receiver and delivers an improvement of 35% for 64KB messages. For larger messages, we see that ‘MV2-GDR-H-Advanced’ peaks out at 5,200 MB/sec while ‘MV2’ and ‘MV2-GDR-H’ deliver higher bandwidth. This is because of the P2P write limitation on Sandy Bridge. Figure 3.12 shows similar trends with bi-directional bandwidth test. Using, ‘MV2-GDR-H-Advanced’, we see a 2x improvement in performance for 4KB messages and a 56% improvement for 64KB messages.

Figure 3.11: Comparison of Bandwidth Performance Using the Existing Host-based Pipelining and the Proposed GDR-based Designs on Sandy Bridge
Collective Performance of Designs using GPUDirect RDMA: The GDR based designs that we proposed have an impact on MPI collective communication operations which are implemented over point-to-point operations. Though 4 processes/nodes is a very small scale, we believe the understanding we gain will help as we scale to large clusters. The existing MVAPICH2 library provides collective specific optimizations for communication from GPU memory. For small messages, it first copies data from GPU onto the CPU, performs the collective operation on the CPU and then copies the result back on the GPU. This reduces the number of CUDA memory copies involved. For large messages, it relies on the host-based pipelining design to hide the copy overheads. MVAPICH2 offers a more advanced design for Alltoall proposed by Singh et. al [82]. It uses a dynamic scheme that does pipelining at the collective algorithm level.

In Figure 3.13, we show that using GDR based schemes considerably reduce the latency of small message MPI_Gather operation. In MVAPICH2, gather operation is implemented with the root posting non-blocking receives from all processes and every other process
posting a blocking send to the root. The non-blocking nature of GDR allows the transfers to progress in parallel and they outperform the latency of CUDA memory copies out of and into GPU memory that happen in the existing implementation of MPI_Gather in MVAPICH2. The GDR based schemes provide 53% improvement in latency when compared to ‘MV2’, for 4Byte messages. As we scale to large number of nodes, this advantage will depend on the efficiency of the concurrency of GDR operations and the structure of the gather tree. We observe that ‘MV2-GDR’, which uses GDR for all message sizes, out-performs ‘MV2’ until 2MByte messages. Beyond that, the P2P bandwidth limitation comes into play. ‘MV2-GDR-H’ benefits from the switch between these two schemes as we move the switching threshold to 2MBytes for MPI_Gather. ‘MV2-GDR-H-Advanced’ shows benefits for messages greater than 128KBytes due to the reduced hop through the host, at the receiver process. As the receiver for all communication is the root process, we see a higher impact of the host bypass in MPI_Gather performance than we had see in point-to-point performance. ‘MV2-GDR-H-Advanced’ shows an improvement of 48%, when compared to ‘MV2’, for 1MByte messages.

**Application level performance with Designs Using GPUDirect RDMA:** We demonstrate the impact of the proposed GDR based techniques on two end-applications: GPULBM and AWP-ODC. GPULBM is a parallel distributed CUDA implementation of Lattice Boltzmann Method (LBM) for multiphase flows with large density ratios [75]. It is an iterative application which operates on 3D Data grids. The decomposition of data is done along the z axis. Hence, the number of elements involved in communication can be obtained as the product of x and y dimensions of the grid times the required 6 degrees of freedom, with each element being of type float. With 256x256x128 data grid, we see a 29% reduction in communication time, a shown in Figure 3.14(a). With 512x512x128 data grid, we
Figure 3.13: Impact of GDR-based Designs on MPI_Gather communication performance
see a 35% reduction, as shown in Figure 3.14(b). The increase in benefits is because of the increase in communication volume. With the 512x512x128 data grid, ‘MV2-GDR-H-Advanced’ improves the evolution time by 8%. Evolution time is the time spent in the main loop of the code which dominates the runtime as the application runs for large number of iterations in a real world run. The application reports this as a measure of the expected performance in real runs. The percentage of communication is expected to increase as we run on larger number of nodes as the decomposition is in the z dimension.

AWP-ODC [97] is a widely used seismic modeling code from San Diego Supercomputer Center and was a Gordon Bell finalist at SuperComputing 2012 [14]. It is a Fortran MPI code. Here, we have used a version of the code written using MPI + CUDA in C and which uses MPI for communication from GPU memory. We present results using two data sets. This version of the code shows a high overhead in data movement between the GPUs. With a 128x128x1024 data set, ‘MV2-GDR-H-Advanced’ improves the communication time in the application by 40% and the overall application run time by 30%, when compared to the ‘MV2’. As in the case of LBMGPU, AWP-ODC reports the time for the main loop which dominates the runtime as the application runs for large number of iterations in real-world [73].

3.4.2 Intranode Communication

We used a 12-core Intel Westmere node with two NVIDIA Tesla C2075s for our experiments. The CPUs are clocked at 2.40 GHz and the node has 24 GB host memory. The node runs Red Hat Linux 5.8, OFED 1.5.3, MVAPICH2-1.8a1p1, and CUDA Toolkit 4.1. We have extended OMB in a couple of ways. We substituted MPI_Wtime with CUDA Event based timers to measure the actual latency of the data movement operations on the
Figure 3.14: Impact on Performance of Lattice Boltzmann Method code

Figure 3.15: Impact on Performance of AWP-ODC code
device. All the experiments presented in this section use two processes and each uses a different GPU device. We first discuss the performance of inter-process communication at the CUDA runtime level. We then compare the performance of the newly proposed designs with the existing alternatives in MVAPICH2, for two-sided communication benchmarks from the OMB suite. Finally, we present the performance of a GPU implementation of Lattice Boltzmann Method (LBM) using the proposed designs.

**CUDA-level performance:** Figures 3.16(a) and 3.16(b) compare the latency of inter-process GPU-to-GPU communication via the host memory with the latency using CUDA IPC. ‘GPU-Host-GPU’ represents the total latency of copying data from device memory of the source process into a shared memory buffer, signaling between the source and destination processes through shared memory and copying the data from the shared memory buffer into the device buffer at the destination process. ‘IPC RemoteGPU-GPU’ represents the latency of copying data from the IPC buffer of a remote process (on the same node) into a local device buffer. ‘IPC GPU-RemoteGPU’ represents the latency of copying data from a local device buffer to the IPC buffer of a remote process (on the same node). For 8Byte messages, both ‘IPC RemoteGPU-GPU’ and ‘IPC GPU-RemoteGPU’ show a 93% improvement in latency over ‘GPU-Host-GPU’. The latency of ‘IPC RemoteGPU-GPU’ differs from that of ‘IPC GPU-RemoteGPU’ in the case of large messages. We believe this is because of how the completion of copies is detected in the CUDA runtime. ‘IPC RemoteGPU-GPU’ and ‘IPC GPU-RemoteGPU’ latencies are 38% and 23% lower than ‘GPU-Host-GPU’, respectively, for 4MByte transfers.

Figure 3.16(c) shows the handle creation and mapping overheads in CUDA IPC. ‘IPC’ is the copy latency from a remote IPC device buffer to a local device buffer. ‘IPC w/ Handle Ovrhd’ includes the ‘IPC’ latency and the overhead of creating and opening the IPC handle.
We observe that the handle overheads outweigh the benefits of the IPC for up to 1MByte. It is important that these overheads are amortized to realize the benefits of CUDA IPC. We handle these overheads using a handle cache inside MVAPICH2 as described earlier.

![Graphs showing performance comparison](image)

(a) Small Message Latency  
(b) Large Message Latency  
(c) Memory Handle Overhead

Figure 3.16: Performance of CUDA IPC

**Point-to-point Communication:** Figures 3.17 and 3.18 compare the latency and bandwidth performance of ‘IPC’, the new designs for two-sided communication using CUDA IPC, with ‘SHARED-MEM’, the existing design in MVAPICH2 that stages GPU-GPU
communication through shared memory. The short message latency of ‘IPC’ can be attributed to the copy from local device buffer to the pre-allocated eager device buffer, the copy from the eager device buffer to the destination buffer, the transfer of packet header and the overhead of calls to cuEventRecord and cuStreamWaitEvent. It avoids the cost of moving data through the host memory, improving the 8Byte message latency by 70% compared to ‘SHARED-MEM’. For larger messages, the cost of staging data through shared host memory increases. ‘IPC’ design removes this overhead and gives close to 79% better latency than ‘SHARED-MEM’ design, for 4MByte messages. The overhead of memory handle creation and mapping is amortized because of the reuse of buffers in the OMB benchmarks. This is representative of most regular applications which reuse communication buffers over their run-time. ‘IPC’ achieves close to 3 times improvement in bandwidth and over 3.5 times improvement in bi-directional bandwidth compared to ‘SHARED-MEM’.

**Impact on Applications:** GPULBM is a parallel distributed CUDA implementation of Lattice Boltzmann Method (LBM) for multiphase flows with large density ratios [75].
GPULBM is an iterative application which operates on 3D Data grids. The decomposition of data is done along the z axis. Hence, the number of elements involved in communication can be obtained as the product of x and y dimensions of the grid times the required 6 degrees of freedom, with each element being of type float. In this section, we compare the performance of GPULBM using two designs for two-sided communication: ‘2SIDED-SHARED-MEM’ which uses staging via host shared memory and ‘2SIDED-IPC’ which uses CUDA IPC. We show the performance of the non-overlapped version here. It is called ‘1SIDED-IPC’. This version uses Put with Post-Wait/Start-Complete synchronization. We see a 16% improvement in the LB step time using ‘2SIDED-IPC’ or ‘1SIDED-IPC’, compared to using ‘2SIDED-SHARED-MEM’ for a 512x512x64 data grid. The message sizes being large, both two-sided and one-sided semantics use direct IPC copies for the transfer. Hence we see that ‘2SIDED-IPC’ and ‘1SIDED-IPC’ perform similarly.
Figure 3.19: Per-step latency in GPULBM. Two processes on an 8-core Intel Westmere node with two NVIDIA C2075 GPUs are used.

3.5 Related Work

Communication latency between host memory and GPU memory is one of the major bottlenecks in a GPU accelerator based system. Several researchers have attempted overlapping memory copy from GPU to host memory with kernel execution on GPU. One such example is the work by Ma et. al. [54]. Jacobsen et. al. [41] have utilized CUDA and MPI interfaces to overlap GPU data transfer and MPI communication with computation to accelerate the computational fluid dynamics simulations. This method has been used in many GPU kernel algorithms optimization, such as the acceleration of a 3D Poisson equation solution in Himeno benchmark [71], and the auto-tuning of the sparse matrix vector multiplications [26]. The authors have used standard MPI interfaces where both MPI and CUDA are unaware of each other. While this results in some performance gain, it is a drain on productivity as it requires careful evaluation and tuning. Moreover, the performance gains are not lasting, i.e. a platform upgrade may change the cost parameters, and the application may need to be re-tuned again. MVAPICH2-GPU provides a simplified method for programmers to not only perform GPU to GPU communication, but with
highest performance. Fan et. al. [32] have proposed Zippy framework to do computation and visualization on GPU cluster. Zippy provides only static communication model and cannot support the advanced communication interface such as collectives. Stuart et. al. [85] have proposed multi-thread framework DGGN on GPU. DGGN still depends on MPI to do the underlying inter node communication. These efforts will benefit from our work as we take care of some of these pipelining internally. Data movement problem on GPU clusters has been investigated in programming models other than MPI. Gelado et.al. propose Asymmetric Distributed Shared Memory model (ADSM) to unify host memory and device memory [35]. The device memory becomes transparent to programmers and the run time system will handle the memory management and data movement through a memory coherence protocol. In this chapter, we focus on the MPI programming model and our approaches work completely in user-space.

MPI intra-node communication on multi-core systems has been studied and optimized by many researchers. Chai, et al. have proposed different shared memory based optimizations for intra-node MPI communication based on the message sizes [24]. Other researchers have shown how kernel-assisted methods can be used to optimize intra-node MPI communication, by reducing the number of copies [23, 46]. Lai, et al. have proposed truly one sided communication on multi-core system based on the kernel assisted method [68]. In this chapter, we extend MPI intra-node communication for heterogeneous clusters with GPUs. Van de Vaart, et al. have demonstrated the potential for using CUDA IPC for large message MPI send/receive communication between GPUs [11]. The performance of GPU-to-GPU communication within a node is affected by many factors such as non-uniform memory access, contention and others as illustrated by Spafford, et al. [84]. It is not easy
task for programmers to manage data movement and synchronization explicitly in applications. Our design can hide these details inside MPI library and provide programmers the standard MPI interface with better performance. We also harness NVIDIA’s GPU Direct RDMA (GDR) feature for MPI libraries and propose hybrid solutions which benefit from the best of both GDR and host-assisted GPU communication.

3.6 Summary

We proposed MVAPICH2-GPU to support efficient GPU-to-GPU communication using MPI. It provides a unified interface for data movement on clusters with NVIDIA GPUs, making it easier to port applications to these clusters. It enables the runtime to transparently optimize communication by using techniques like pipelining and by taking advantage of advanced features like CUDA IPC and GPUDirect RDMA. We have shown the benefits of our designs on point-to-point and collectives communication using micro-benchmarks, application kernels and end applications. MVAPICH2-GPU can significantly improve the performance even when compared with the optimized codes that overlap data movement between GPU and the Host with data movement over the network. We show up to 69% and 45% improvement in point-to-point latency for data movement for 4Byte and 4MB messages, respectively. Likewise, the solutions improve the bandwidth by 2x and 56% for 4KByte and 64 KByte messages, respectively. We observed similar improvements for bi-directional bandwidth. Our results show an improvement of 30% in the overall application execution time of AWP-ODC and up to 35% reduction in the communication time of GPULBM.
Chapter 4: Extending OpenSHMEM for GPU Computing

PGAS models like OpenSHMEM, with better programmability and high-performance one-sided communication, present a good case for developing applications on GPU clusters. However, the current standards do not provide efficient integration with GPU programming frameworks like CUDA and OpenCL. For example, in current OpenSHMEM applications that involve data movement between GPUs, the developer has to separately manage the data movement between GPU device memory and main memory at each process, using CUDA or OpenCL, as well as the data movement between processes, using OpenSHMEM. Figure 4.1(a) shows the various operations that will have to be performed in order to achieve a simple `shmem_putmem` operation from GPU memory to GPU memory. The programming overhead incurred is beyond the number of lines of code shown in Figure 4.1(a) as one has to employ intricate pipelining designs and take advantage of advanced features like CUDA Inter Process Communication (IPC) to achieve optimal GPU-GPU communication performance. Moreover, the selection and tuning of these designs is not straightforward on the modern NUMA and multi-GPU node architectures. The inability of the OpenSHMEM model to support GPU-GPU communication also prevents library implementations from offering such optimizations transparently to the user. As shown in
Figure 4.1(a), the current model also nullifies the benefits of asynchronous one-sided communication by requiring the target process is to perform a CUDA memory copy in order to complete the transfer.

| PE 0 | cudamempy (host_buf, device_buf, . . . )  
|      | shmemp_putmem (host_buf, host_buf, size, pe)  
|      | shmemp_barrier ( . . . )  
| PE 1 | shmemp_barrier ( . . . )  
|      | cudamempy (device_buffer, host_buffer, size, . . . )  

(a) Current OpenSHMEM model

| PE 0 | shmemp_putmem (device_buf, device_buf, size, pe)  
| PE 1 | – no operations required –  

(b) Enhanced OpenSHMEM model (presented in this chapter)

Figure 4.1: Programming with current and enhanced OpenSHMEM Models

In this section, we present extensions that enable applications developers to use OpenSHMEM for communication directly from GPU memory while preserving the one-sided semantics. The code snippet using the proposed model is shown in Figure 4.1(b) We extend the OpenSHMEM memory model in such a way that it is compatible for use with CUDA and OpenCL, two of the most widely used programming frameworks for GPUs. The extensions also enable the OpenSHMEM runtime to significantly optimize communication for
different GPU node configuration. We demonstrate the benefits of the proposed model and runtime using micro-benchmarks and application kernels.

4.1 Limitations of Existing Alternatives

Several researchers have proposed extensions to PGAS models for use on GPU clusters [49, 52, 98]. In this section, we analyze some of these proposed extensions in the context of OpenSHMEM and explain their limitations.

![Figure 4.2: Existing Heap Selection Based Designs](image)

In PGAS programming models, each process contributes a memory segment to form a shared global address space. In the context of UPC, Zheng et al. have proposed an extension to specify where this shared memory segment resides for a given UPC thread [98]. The memory can reside either on the host or on the GPU, but not both. Following a similar approach in OpenSHMEM restricts the location of symmetric heap at a given process to the host or the GPU as depicted in Figure 4.2(a). The buffers returned by allocation routines like shmalloc, shmemalign and others will be GPU device buffers when heap is specified to be on the GPU. This requires minimum or no extensions to the API as the selection of
heap location can be provided using an OpenSHMEM environment variable. While this works for applications where processes compute and communicate exclusively from GPU memory, it does not yield well for applications that involve both host and GPU buffers. Developers can work around this using Multiple Process Multiple Data paradigm, but this severely restricts the general programmability. We refer to this approach as "Static Heap Selection".

Miao et al. have proposed a multi-domain model for UPC [52], where the application developer can dynamically select the domain (the host or a GPU) where the shared memory region is allocated. A similar extension can be applied to OpenSHMEM by allowing dynamic selection of the heap. Figure 4.2(b) depicts the dynamic heap model with two independent heaps, one for the host and one for the GPU. Although this approach addresses the programmability issue seen with the static heap approach, there exist several interoperability issues with CUDA and OpenCL that this model does not consider. We discuss them in detail below.

**Device Buffer Allocation and Detection:** Memory allocation routines in OpenSHMEM return the address of the allocated memory buffer to the user. All data transfer routines take the source and destination addresses as input parameters and move data between them. This works well with CUDA which also uses addresses to represent device memory. The pointer to a device buffer returned by an OpenSHMEM allocation routine can be used directly in CUDA calls. However, when an address is given as input to an OpenSHMEM data movement routine, the OpenSHMEM runtime should be able to detect if the buffer it points to is on the host or on the GPU. CUDA versions 4.0 and later offer a feature known as Unified Virtual Addressing (UVA) which provides a single address space for the host and the GPUs. Using a CUDA function call - `cudaPointerGetAttributes`, users can detect if
a given address points to a host buffer or a device buffer. This can be used by the OpenSHMEM runtime to detect device buffers and handle data transfers appropriately. In versions of CUDA without UVA, there is no way for a runtime to perform such differentiation as the same virtual address can exist on both host and GPU address spaces.

In the OpenCL framework, buffers are represented as memory objects. Data movement routines in OpenCL require use of objects and corresponding offsets to indicate source and destination buffer locations for the transfer. When a GPU device buffer is allocated using an OpenSHMEM allocation call, the OpenSHMEM runtime requires a memory object to virtual address conversion, which it then returns to the user. A virtual address to object conversion routine should also be provided for the user to flexibly use the buffer in OpenCL calls. Since OpenCL does not offer a feature like UVA, the OpenSHMEM runtime needs a mechanism to differentiate between the virtual addresses associated with device buffers and host buffers. To be interoperable with different versions of CUDA and OpenCL, one needs to take this into consideration while extending the OpenSHMEM model.

**Context Management:** A context encapsulates all the GPU activity and resources owned by a process. When using the CUDA runtime API (prefixed with cuda), contexts are managed implicitly. A context is created for each device during the first runtime API call. However, this information is not exposed to the user. Hence explicit context handling by the user or the OpenSHMEM runtime is not required. CUDA driver APIs allow advanced users to explicitly create and manage contexts. Only one context can be active for a process, at any given point of time. When a context is made current, the following CUDA calls are executed in that particular context. In this scenario, the user will have to select the appropriate context before calling any OpenSHMEM calls that allocate or act on a device buffer
and any CUDA calls made by the OpenSHMEM runtime will automatically be executed in that context.

On the other hand, context management is always explicit in OpenCL. Context is a required parameter in many OpenCL API calls including calls for buffer allocation and data movement. Consequently, if OpenSHMEM APIs are to be extended to allocate device buffers for communication, there has to be a way for the user to pass the corresponding context to the OpenSHMEM runtime. The models mentioned above do not consider these constraints and hence are not interoperable with OpenCL.

### 4.2 Extending the OpenSHMEM Memory Model

![Symmetric Heap Model: Mapping device buffer objects](image)

Figure 4.3: Symmetric Heap Model: Mapping device buffer objects
We propose the symmetric map model to handle the limitations outlined above. We believe that several of the interoperability issues arise from the fact that OpenSHMEM manages allocation of communication buffers. Our proposed approach allows more flexibility and minimizes interoperability issues by allowing the users to manage GPU device buffer allocation while allowing them to map the buffers into a symmetric address space for communication.

**Symmetric Map:** Symmetric Map is a segment of virtual address space reserved at each process, onto which GPU buffers can be mapped using a collective `shmap` call. The call returns an address pointer (from the map) which can be used in any future OpenSHMEM communication calls. The map pointer cannot be used for direct access using CUDA/OpenCL calls. However, the user is free to use the original device buffer in any CUDA/OpenCL calls. A device buffer has to be unmapped using a `shunmap` before it is freed.

The `obj` parameter is the address containing a memory object in the case of OpenCL or a pointer to the device buffer in the case of CUDA. The size of the device buffer is specified using the `size` parameter. The `obj_type` parameter tells the OpenSHMEM runtime the type of object contained in `obj`. We support two types of objects in our design: `OSHM_MEMTYPE_CUDA` and `OSHM_MEMTYPE_OPENCL`. The `shmap` calls follow semantics similar to the standard memory allocation routines in that every

```c
void *shmap (void *obj, size_t size, int obj_type);
void shunmap (void *ptr);
```
process is required to make the call with the same buffer size at the same point of execution in the program. Figure 4.3 shows the state of heap before and after a *shmap* call on multiple PEs. This symmetric mapping and the information about the symmetric address space that is exchanged during runtime initialization allows for a direct translation between a local map address and a remote map address. The symmetric map provides a way to translate between memory objects and address pointers that can be used under the hood for other OpenSHMEM API calls. Now we look at how this model handles the interoperability issues outlined above and how symmetric map avoids the need for further changes to the OpenSHMEM API.

As the symmetric map is part of the virtual address space reserved during OpenSHMEM runtime initialization, a simple pointer check can decide if a pointer specified in an OpenSHMEM call is a host buffer or is a device buffer that was mapped using *shmap*. One option for implementing a symmetric map is to use the *mmap* POSIX call with the `MAP_ANONYMOUS` option. The symmetric map does not require any physical pages to be reserved and hence does not consume any physical host memory. The runtime translates the mapped address into a device memory address (for CUDA) or a memory object (OpenCL) before issuing any operations on it. Translation of a remote mapped address can be done in several ways based on features available on a given GPU hardware.

We have outlined the issues with context management when operating with OpenCL in Section 4.1. Since the device buffer is allocated by the application developer, they have complete control over it and any associated context information. Since the *shmap* call provides the OpenSHMEM runtime access to the memory objects, it can query the context of a given buffer object in OpenCL using the `clGetMemObjectInfo` call. The symmetric
heap model, by allowing the user to manage device buffer allocation and management, minimizes the interoperability constraints between CUDA/OpenCL and OpenSHMEM. All the standard API calls in OpenSHMEM remain unchanged ensuring backward compatibility of the extended standard.

### 4.3 Symmetric Heap and Address Translation

Virtual address space required for the symmetric map is reserved using the `mmap` call during OpenSHMEM runtime initialization, as outlined in Section 4.2. We have extended the memory allocator module in the OpenSHMEM reference implementation [66] to manage this address space as objects are mapped and unmapped. The amount of address space reserved and hence the amount of device memory that can be mapped is controlled by a runtime variable: `OSHMSYMMETRICMAPSIZE`. The base pointers of the symmetric map are exchanged between processes for future address translations.

**Using CUDA:** CUDA versions 4.0 and later offer a feature called Inter Process Communication (IPC) that allows one process to map the GPU device buffer of another process running on the same node into its address space. The remote buffer can then be directly accessed using CUDA data movement calls and kernels. When IPC is available between GPUs used by two processes, they create and exchange IPC handles during the collective `shmap` call. These handles are mapped and the resulting addresses are stored along with the symmetric map of the local buffer. When address translation occurs during an OpenSHMEM call, the runtime can immediately convert a local address into the mapped IPC address of the remote buffer (by virtue of symmetric mapping), that it can directly use in CUDA calls for data movement.
However, on current Intel Architectures, CUDA IPC is available only between GPUs on the same I/O Hub (IOH). We work around this limitation using a design we refer to as "shadow context." For example, consider two processes P0 and P1 using GPUs G0 and G1 that are connected to different IOHs. During OpenSHMEM runtime initialization each process creates a context (shadow context) on the other process’s GPU. When an \textit{shmap} is called with a CUDA device buffer, the process makes the shadow context active using the CUDA context management routines and maps the remote process’s buffer into its address space using CUDA IPC calls. The mapped buffer is stored along with the symmetric map of the local buffer. When an OpenSHMEM data movement operation is called, the runtime translates the local address into the corresponding address that was mapped using IPC. It then uses CUDA Peer-to-Peer (P2P) transfers to move data from the remote GPU to the local GPU. This is explained further in Section 4.4.

When processes are running on different nodes, the map information is stored at each process, locally. Although a translation from local to remote addresses is straight forward, the actual data transfer requires intervention from the OpenSHMEM process or an associated service thread on the remote node. Thus, we avoid storing global mapping information at each process. This is discussed in detail in Section 4.4. The GPU Direct RDMA feature available with CUDA 5.0 has made direct access feasible when appropriate support is provided by the network card vendors. In such a case, the mapping can be exchanged and stored at each process during the \textit{shmap} call.

\textbf{Using OpenCL:} In the case of OpenCL, memory objects are passed to the runtime in the \textit{shmap} call. The runtime retrieves the context associated with the buffer. If the context is new, the runtime creates any required resources like command queues that are later used.
for asynchronous data transfers. These contexts and the queues are cached for quick access. The required symmetric map address space is reserved for the object and the mapped pointer is returned to the caller. As in the case of inter-node transfers with CUDA, the mapping information is stored locally at each process.

4.4 Designs for an Efficient Communication Runtime

In this section, we discuss the design and implementation details of the various data movement APIs in OpenSHMEM. We take advantage of the different communication channels available to efficiently implement these operations.

4.4.1 Intra-node Communication

We present designs for intra-node communication when GPUs are connected in intra-IOH (intra-Socket) and inter-IOH (inter-Socket) configurations that will ensure true asynchronous progress for one-sided communication in OpenSHMEM. The designs are presented in the context of CUDA. OpenCL does not offer a feature like CUDA IPC that is provided by CUDA. Hence, we use a host-based pipeline design for both intra-node and inter-node communication with OpenCL. The only difference between the two is use of shared memory copies and the use of RDMA transfers. We discuss this design in the following section for inter-node communication.

**Intra-IOH Communication with CUDA**: As outlined in Section 4.3, CUDA IPC allows remote GPU buffers to be mapped on to the address space of the local process. With this feature, the communication operations can be implemented directly over `cudaMemcpy` calls. CUDA IPC is capable of transferring data directly over the IOH, removing the need for a copy into host memory. Hence, this provides the fastest method of data movement.
between GPUs. Figure 4.5 depicts the difference between IPC-based and host-based transfers. The direct memory access model offered by IPC enables a true one-sided implementation of OpenSHMEM put/get routines. The strided put/get operations are implemented over `cudaMemcpy2D`. The ordering of blocking IPC memory copies and any following local CUDA operations is ensured by the runtime. However, their ordering with CUDA operations at the remote process needs to be explicitly controlled through IPC events. Pair-wise IPC events are created and exchanged between processes during runtime initialization phase. After an IPC memory copy, the process records the remote process’s IPC event using a `cudaEventRecord` call. The remote process has to call `cudaStreamWaitEvent` on its event to ensure ordering of the copy and any following CUDA operations on the buffer. It is to be noted that this is done during OpenSHMEM synchronization routines and hence does not affect the one-sided nature of the data transfers.

![Figure 4.5: Intra-Node communication using CUDA: Intra-IOH and Inter-IOH Configurations](image)
**Inter-IOH Communication using CUDA:** The `cudaMemcpyPeer` call copies data from one GPU device to another and works in both Intra-IOH and Inter-IOH configurations. This coupled with the shadow context described in Section 4.3, enables us to implement inter-IOH data transfers in a truly one-sided manner. It also reduces the overhead of explicitly staging data through the host buffers and thus provides better performance. However, `cudaMemcpyPeer` calls are serialized with respect to other activities on the GPU and can affect any computation and communication overlap [27]. We overcome this by using `cudaMemcpyPeerAsync` call on a dedicated stream which makes it asynchronous with respect to all the work on other streams and devices. We then poll for its completion using a CUDA event recorded on that stream.

### 4.4.2 Inter-node Communication

In this section, we present design for efficient inter-node communication between GPUs which uses a helper thread to ensure true asynchronous progress of one-sided communication. Designs using both CUDA and OpenCL are discussed.

**Inter-node Communication using CUDA:** For inter-node GPU-to-GPU communication, the data has to be moved from the source GPU onto the source host, transferred onto the remote host and then copied into the remote GPU. We use the approach of staging data through the host. The three stages: Device-to-Host, Host-to-Host and Host-to-Device can be pipelined in the presence of networks like InfiniBand to completely hide the staging cost. The pipeline block size can be tuned based on the cost model of CUDA memory copies and InfiniBand RDMA transfers. In Section 3.2.1, we had proposed this design in the context of MPI. Similarly, the "GPUDirect-RDMA" feature discussed in Section 3.2.2 can be applied...
here, in the context of OpenSHMEM. In this report, we focus on pipeline-based designs for OpenSHMEM.

In our earlier approach, we had used CUDA streams to handle the pipeline of CUDA memory copies. In this design, we use CUDA events, that are light-weight when compared to streams and take up fewer GPU resources. We use one stream for Device-to-Host and Host-to-Device transfers to avoid unnecessary ordering between copies and to take advantage of the bi-directional PCIe bandwidth. To ensure true one-sided progress for inter-node data movement, we enable the service thread offered by the OpenSHMEM reference implementation. Before the pipeline executions starts, there is a control message exchange during which the address translation happens on the remote process and host buffers are reserved for the transfer. In the case of strided transfers, we pack data on the device using `cudaMemcpy2D` before moving the data out onto the host. This avoids significant overheads incurred by `cudaMemcpy2D` operations from device to the host.

![Figure 4.6: Inter-node communication using CUDA or OpenCL](image)

Figure 4.6: Inter-node communication using CUDA or OpenCL
Communication Using OpenCL: Data movement between GPUs with OpenCL is implemented by staging it through host memory. We have implemented a pipelined design similar to the one with CUDA. OpenCL offers command queues, which are analogous to streams in CUDA, and events that can be used to test for completion of asynchronous copies. Similar to our design with CUDA, we adjust the pipeline block size to achieve maximum overlap and performance.

4.4.3 Atomics and Synchronization

In this section, we present designs for efficient implementation of synchronization and atomic operations in OpenSHMEM.

Atomics: Both CUDA and OpenCL frameworks offer atomic operations on device memory. In this work, we take advantage of these in the case of CUDA by using kernels to implement OpenSHMEM atomics. A similar approach can be applied with OpenCL as well. This avoids the need to explicitly move the data on to the CPU, perform the atomic and copy it back, all in a critical section. When CUDA IPC is available, the kernel can be executed directly on the remote device buffer. In other cases, a message is sent to the remote process and the associated service thread executes the kernel. When the atomic has a return value we use device pointers of mapped host buffers (returned by `cudaHostGetDevicePointer` call) for the kernel to copy the value to the host rather than using an explicit memory copy. This allows the CUDA runtime to overlap the data movement with the atomic operation.

Synchronization: OpenSHMEM provides synchronization routines that ensure ordering and completion of communication operations. We consider the implementation of `shmem_barrier_all` and `shmem_barrier` calls which are collective operations on all the
processes and any subset of processes, respectively. These calls ensure the completion of previously issued OpenSHMEM operations in memory of local and target processes. When CUDA IPC is used, each process has to wait for all issued IPC event record calls to complete before returning from a barrier call. We do this efficiently using *cudaStreamWaitEvent*, that performs synchronization on the device. In the case of inter-node communication with both CUDA and OpenCL, each process ensures that any incoming data movement requests and the corresponding memory copies onto the device are complete before returning from the barrier call. The ordering of barrier messages and any earlier communication requests is ensured by the ordering guarantees of the underlying shared memory and network channels.

### 4.5 Redesigning Application Kernels

We have considered two applications benchmarks: Stencil2D and BFS, from the Scalable Heterogeneous Computing (SHOC) Benchmark Suite to evaluate the benefits of the extended OpenSHMEM model and the efficient runtime on application performance. We have modified the CUDA+MPI version of Stencil2D benchmark to use `shmem_putmem` instead of `MPI_Irecv`/`MPI_Send` for communication. The existing version of the benchmark moves data from GPU onto the host using `cudaMemcpy` and `cudaMemcpy2D` before exchanging it between the processes. We follow a similar approach in the *Naive* version. In the *Enhanced* version we use OpenSHMEM operations for direct communication from the GPU buffers. We use `shmem_putmem` operations to exchange contiguous north/south boundaries and `shmem_float_iput` to exchange non-contiguous east and west boundaries. The computation part of the application benchmark is untouched and is similar for both *Naive* and *Enhanced* versions.
All the kernels in the SHOC benchmark suite except Stencil2D were designed to be run on a single GPU. We have developed a task parallel version of the default BFS algorithm in SHOC using OpenSHMEM by linearly distributing the vertices evenly across processes. During graph traversal, each vertex maintains (sendlists) of vertices it has discovered but are owned by remote processes. At the end of each step, the processes exchange this information before moving onto the next step of the traversal. Each process marks the unvisited vertices in the receivelist for traversal in the next iteration. When there are no new vertices discovered at the processes, the traversal ends. This design is exactly the same for both Naive and Enhanced versions of the code. The only difference is that in Naive version, we copy the list of remote vertices onto the host and write to the receivelist on the host at the corresponding process, using OpenSHMEM put operations. Then, we copy the receivelist onto the GPU and continue to the next step of the traversal. In the Enhanced version, we issue OpenSHMEM put operations directly on from sendlists on the local device to the receivelist on the remote device. The offset into the receivelist in both schemes is managed using a host counter updated using atomics.

4.6 Experimental Evaluation

We have used two clusters in our experiments.

**Cluster-A:** It consists of two nodes. Each node is equipped with two Intel Xeon E5645 quad-core CPUs operating at 2.40 GHz, 12GB of main memory, a Mellanox MT26428 QDR HCA and two NVIDIA Tesla C2075 GPUs that have 5GB of memory. The nodes have Red Hat Linux 5.4, OFED 1.5.1 and CUDA 4.1. All the micro benchmark evaluation was done on these nodes.
**Cluster-B:** This is the XSEDE Keeneland-KIDS cluster where each node is equipped with two Intel Xeon X5560 six-core CPUs, 32 GB of main memory, a Mellanox MT4099 IB FDR HCA and three NVIDIA Tesla M2090 GPUs. The application benchmark level evaluation is carried out on this cluster.

**Data Movement:** Figures 4.7 and 4.8 compare the performance of *Enhanced* and *Naive* for intra-node communication with CUDA. As discussed in Section 4.4.1, IPC-based design for intra-IOH transfers delivers the best performance for GPU-GPU communication. For 4 Byte transfers it provides a 90% improvement in latency of both put and get operations when compared to *Naive*. We observe lower latencies with put operation than with get operation because the completion semantics of put do not require the target buffer to be updated when the call returns. For 4 MByte transfer, *Enhanced* gives an improvement of 67% and 72% in the latencies of put and get operations, respectively.

We show that using *cudaMemcpyPeer* reduces the latency compared to host-based transfers for inter-IOH configuration. It provides a single-copy direct memory access semantics and also allows the underlying CUDA runtime to optimize the transfers. Using this design, *Enhanced* provides 22% and 40% improvement in put and get latencies, respectively, for 4 Byte transfers. It shows a 43% and 45% improvement in the put and get latencies, respectively, for 4 MByte transfers.

Figures 4.9(a) and 4.9(b) present the performance of inter-node OpenSHMEM get operation using the CUDA platform. Registering host buffers with the CUDA runtime has an impact on the memory copy latencies observed by applications. It improves the performance of large copies as it allows the runtime to take advantage of the DMA. However, we have observed that registration has a negative impact on small copy latencies. Hence,
Figure 4.7: Intranode \textit{shmem\_putmem} Performance with CUDA

Figure 4.8: Intranode \textit{shmem\_getmem} Performance with CUDA
we selectively register the buffer pools used for large message transfers in the OpenSHMEM runtime. This results in a 17% improvement in latency for 4 Byte transfers using Enhanced, when compared to Naive version. In the Naive version, the host buffers used in the benchmark are always registered with the CUDA runtime. The pipelined design in Enhanced delivers a 42% improvement in get latency for 4 MBytes transfers, when compared to the Naive design. The pipeline block size tuned for cluster A is 128 KBytes.

Figure 4.9(c) shows impact of the OpenSHMEM service thread in providing asynchronous progress for one-sided inter-node communication. Enhanced with the service thread provides constant get latency irrespective of the skew (a busy compute loop in the main thread) present at the remote process. Naive version, which depends on benchmark level calls at the remote process for the GPU-GPU transfer to complete, shows a degraded performance as the skew increases.

With OpenCL, we observe similar latencies for small inter-node OpenSHMEM put transfers with both Enhanced and Naive versions. For large messages, pipelined design in Enhanced shows up to a 42% improvement in performance compared to the Naive version, as shown in Figure 4.10. The tuned pipeline blocksize is 128 KBytes for OpenCL based communication as well.

Atomics: Our extended model allows OpenSHMEM atomics to be executed directly on remote GPU buffers. It also allows the runtime to optimize these using GPU kernels as outlined in Section 4.4.3. In intra-IOH configuration, Enhanced shows 50%, 55% and 50% improvements with integer Fetch-and-Add, Swap and Compare-and-Swap operations respectively, compared to Naive. In inter-node configuration, we see improvements of 51%, 52% and 51% improvement respectively for the same operations.
Figure 4.9: Internode *shmem_getmem* Performance with CUDA
Application Benchmark Evaluation: Figures 4.12(a) and 4.12(b) present a comparison between the *Naive* and *Enhanced* versions of the Stencil2D kernel of the SHOC benchmark suite. On 192 GPUs, we see consistent performance gains with the *Enhanced* version even as the problem size increases. With a 4Kx4K problem size per GPU, we see a 65% improvement in performance on 192 GPUs. As outlined in Section 4.4.2, the OpenSHMEM runtime optimizes the strided operations by packing data on the device. The original and the *Naive* version of the benchmarks use *cudaMemcpy2D* to directly move data from the GPU to the host. This attributes to the large gains in performance using the optimized version.

We have parallelized the BFS kernel of the SHOC benchmark suite to run on multiple GPUs. Figure 4.12(c) presents a performance comparison between the *Naive* and *Enhanced* versions of the benchmark. These versions differ only in the way OpenSHMEM is used for GPU-GPU communication making the rest of the code performance neutral, as
described in Section 4.5. With 1 million vertices per GPU, we see up to a 12% improvement in performance on 96 GPUs. We also see a consistent improvement in performance as we scale from 24 to 96 GPUs.

![Graphs showing performance comparison](image)

(a) IntraIOH  
(b) Internode

Figure 4.11: OpenSHMEM Atomics Performance with CUDA

### 4.7 Related Work

In recent years, a fair number of research efforts have been dedicated to eliminate the programming complexity from the separated memory address space of the GPU device memory and the CPU host memory. A compiler based method has been proposed to unify the address spaces for GPU device memory and the CPU host memory [34, 91]. In these cases, the runtime system handles the data consistency between the device and the host. However, compiler based approaches are hard to extend to clusters with multiple nodes where a distributed memory programming model, such as MPI, is required. Many researchers have proposed application-centric interfaces [56, 79] to unify the address spaces...
Figure 4.12: SHOC Application Benchmark Performance with CUDA on up to 192 GPUs (3 GPUs/node)
and automatically extend the applications on GPU clusters. This method is specific to one type of the application, such as stencil-based application. Moreover, their run time systems depend on distributed memory programming models which decide the performance of the application.

Several designs have been proposed to alleviate the burden of data movement and buffer management between different address spaces in the existing programming models, such as MPI and PGAS. We discussed MVAPICH2-GPU in Chapter 3, which allows GPU to GPU data communication using the standard MPI interfaces. We presented optimizations inside MPI runtimes to optimize inter-node communication by overlapping data movement over the PCIe with RDMA transfers over the InfiniBand network. For the intra-node communication, CUDA IPC-based optimizations have been proposed for the intra IOH data movement and shared memory based pipeline is used for the inter IOH data movement. Similar optimizations have been presented in other popular open source MPI libraries such as MPICH2 and OpenMPI [42–44]. For the UPC programming model, researchers have proposed extensions to the UPC compiler to support communication from GPU device memory and take advantage of GPU clusters [25, 98]. Designs to optimize the UPC runtime for GPU-GPU communication have also been proposed [52]. This chapter focuses on extending the OpenSHMEM programming model to support communication from GPU device buffers, while ensuring its interoperability with both CUDA and OpenCL computing frameworks. We present a novel design called shadow contexts to use peer-to-peer copies for inter-IOH communication. We enable true one-sided communication using helper threads for efficient internode transfers. To the best of our knowledge, this is the first work that enables GPU-GPU communication using the OpenSHMEM programming model.
4.8 Summary

In this chapter, we have extended the OpenSHMEM PGAS model to support communication from GPU device memory. This work, while improving programmability of GPU applications using OpenSHMEM, allows an efficient runtime to provide optimal performance in various GPU configurations in a transparent manner to the user. We also ensure that the proposed extensions are interoperable with the two most popular GPU programming frameworks: CUDA and OpenCL. To the best of our knowledge, this is the first work that enables GPU-GPU communication using the OpenSHMEM model, for both CUDA and OpenCL computing frameworks. The proposed extensions, coupled with an efficient runtime, improve the latency of 4 Byte \texttt{shmem\_getmem} latency by 90\%, 40\% and 17\%, for intra-IOH, inter-IOH and inter-node GPU configurations with CUDA, respectively. For these configurations, the proposed enhancements improve the latency of 4 MByte \texttt{shmem\_getmem} operation by 72\%, 45\% and 42\%, respectively. We see similar improvements in the latencies of \texttt{shmem\_put} operation. The proposed runtime improves the performance of OpenSHMEM atomics by up to 55\% and 52\%, for intra-IOH and inter-node GPU configurations, respectively. The proposed enhancements have shown a considerable impact on the performance of application benchmarks. They improve the performance of Stencil2D and BFS kernels by 65\% and 12\% on clusters of 192 and 96 GPUs respectively.
In the earlier chapters, we have discussed programming model extensions and runtime level designs to optimize data movement on GPU clusters. As discussed in Section 2.4, Intel MIC architecture provides an attractive alternative to GPUs. It allows a variety of ways in which many-core processors can be used. The most flexible one is the symmetric mode when there are MPI processes running on both the host and on the coprocessor. The number of processes and the degree of threading can be decided by the application developer based on the use case. When there are MPI processes running on both the host and the coprocessor, there are several types of communication that can happen between the processes: MIC-to-MIC, MIC-to-Host, Host-to-MIC and Host-to-Host. Each of these can be intra-node or inter-node. In this section, we discuss designs in MVAPICH2 to optimize MPI communication on Xeon Phi clusters, by taking advantage of all the transports available.

5.1 Intra-MIC Communication

Xeon Phi supports POSIX shared memory API for inter-process communication within the coprocessor. MVAPICH2 uses shared memory as the default channel for intra-node communication and can be used out-of-the-box for running MPI jobs on the MIC. When
there is a message to be exchanged between two MPI processes, the sender process copies a header and the data into a shared memory region allocated during MPI_Init and the receiver process, that keeps polling for any incoming messages, detects the header and copies the data into the destination buffer. For large message exchanges, it is natural for this transfer to be pipelined where the sender copies chunks of the message into the shared memory region and the receiver copies out the chunks as they are copied. MVAPICH2 uses a chunk size of 8KB on most host architectures and this is the default value. The Xeon Phi can choose between a multi-threaded memory copy or a DMA-assisted copy to transfer data in a `memcpy` operation. As DMA offers lower latency compared to the multi-threaded copy for large messages, we have tuned the copy chunk size in MVAPICH2 to the DMA copy threshold (64KB from our experiments) in order to take advantage of the low latency DMA-assisted copies.

The SCIF interface, a part of the MPSS stack, provides direct control for the user to choose when the DMA engine should be used for a data transfer. From our experimentation, SCIF using DMA incurs lower overhead compared to memory copies and hence allows the MPI library to achieve lower latency and higher bandwidth. Hence, we extend the MVAPICH2 SMP channel to take advantage of SCIF for large messages. We establish SCIF connections between process pairs running on the same MIC. During a large message transfer, we use the `RPUT` [87] rendezvous protocol where the receiver process registers and sends the SCIF offset of the destination buffer to the sender process. These control messages are exchanged using the default SHM channel. The sender process then uses `scif_writeto` to transfer data from the source buffer to the destination buffer. The two processes asynchronously poll for completion of the transfer using `scif_fence_signal` call as discussed in Section 2.4. Registration and de-registration of buffers with SCIF can add
a significant overheads if done in the critical data transfer path. To avoid this, we have incorporated SCIF registration into the registration cache in MVAPICH2 which caches registered entries and does a lazy de-registration by trapping any memory free calls.

![Communication Channels within a MIC in MVAPICH2](image)

Figure 5.1: Communication Channels within a MIC in MVAPICH2

### 5.2 Intra-node Communication Between MIC and Host

As discussed in Section 2.4, MPSS on Xeon Phi offers support for the OFED IB verbs interface for communication between MIC and host or between two MICs, both within a node and across nodes. The direct IB verbs interface provided offers low latency but has to depend on the InfiniBand adapter even to move data between a MIC and the local host. The bandwidth in this channel is limited by the peer-to-peer bandwidth available between MIC and the IB HCA on a given platform and the bandwidth offered by the IB HCA itself. The IB-SCIF interface (which is an implementation over PCIe) can also be used for communication between MIC and host. However, it incurs a high latency as it does not provide a kernel bypass [17]. However, it provides higher bandwidth for communication between
MIC and host within a node, when compared to native IB. MVAPICH2 can directly support communication on an Intel MIC cluster using the IB or IB-SCIF interface as shown in Figure 5.2(a). But, to avoid the overhead incurred by the IB-SCIF’s verbs implementation over SCIF and the limitations in peer-to-peer bandwidth seen with the IB HCA, we design a direct channel extension, SCIF-CH3, to implement large message communication between MIC and Host as depicted by ‘SCIF’ path in Figure 5.2(a). The implementation of the SCIF-CH3 channel is similar to the SCIF based channel for Intra-MIC communication. To take advantage of the low latency offered by IB and the high bandwidth offered by SCIF, we provide a hybrid design by selecting the right channel based on the size of message being transferred. We use the direct IB interface (when an IB adapter is available) to exchange the control messages and small data messages to exploit the low latency path. We use the DMA-assisted data movement offered by SCIF for all large message transfers, thus achieving higher bandwidth.

**Host-Initiated SCIF:** The DMA-assisted transfers using SCIF can be initiated on the MIC or the host. However, host initiated DMA has lower overhead compared to that issued from the coprocessor. To take advantage of this behavior, we have implemented a Host-Initiated SCIF design where the transfers from MIC to host are implemented using the *RGET [87]* rendezvous protocol where the process on the host initiates a *scif_readfrom*, depicted in Figure 5.2(b). However, issuing all the DMAs from the host may result in unused resources on the coprocessor. To achieve maximum bi-directional bandwidth, for example, one will have to issue DMAs from both coprocessor and the host. Taking this into consideration, for collective operations like Alltoall, where bi-directional bandwidth is critical for large messages, we provide the flexibility to initiate DMA transfers from both the coprocessor
and the host. In communication patterns that are latency critical or that depend on unidirectional transfers (rooted collectives for example), we allow the users to take advantage of host-initiated transfers.

![Diagram of MIC-Host Communication Channels and Host Initiated Transfers](image)

(a) MIC-Host Communication Channels  
(b) Host Initiated Transfers

**Figure 5.2: Communication Between Host and MIC**

### 5.3 PRISM for Internode Communication

In this section, we discuss MVAPICH-PRISM, a PRoxy-based communication framework using InfiniBand and SCIF for Intel MIC clusters, for improving the performance of inter-node MPI communication operations. The framework efficiently takes advantage of both IB and PCIe channels. It includes a design using Direct IB and multiple designs using host proxy to optimize the collection of communication paths possible in the symmetric mode.
5.3.1 Design Considerations

For the rest of this chapter, we refer to processes located on the MIC as a *mic-process* and processes on the host processor as a *host-process*. The MPSS software stack allows access to the IB HCA from inside the MIC coprocessors through its CCL (Coprocessor Communication Link) driver. We refer to this as DirectIB. As MIC is connected to the host as a PCIe device, the data movement between MIC and the IB HCA is carried out as peer-to-peer communication over the PCIe bus. As highlighted in Section 1, current generation Sandy Bridge processor chipsets from Intel provide very limited peak bandwidth for peer-to-peer communication, especially when reads from MIC memory are involved. The InfiniBand HCA directly issues memory read/write operations when data needs to be exchanged with remote MIC or remote Host. Owing to the hardware limitation, this leads to very low communication performance. This issue has to be addressed in order for communication libraries, and thus applications, to deliver good performance on clusters with MIC. The peer-to-peer communication limitations become even more evident when the IB HCA and the Xeon Phi are connected to different I/O Hubs (IOHs) or sockets. As clusters with multi-rail IB network and MICs are deployed, this scenario will also have to be considered. It is not clear as to which parts of these problems will be resolved in next generation of chipsets.

The MPSS stack provides the SCIF interface that implements communication over the PCIe bus, between a MIC and its Host. SCIF has been shown to deliver close to peak PCIe bandwidth performance through its RMA (Remote Memory Access) semantics. One way to work around the limitations with DirectIB is to stage data through the host, by taking advantage of the SCIF and host-to-host IB channel. However, if the process resides on the
MIC, we need an agent/proxy on the host to stage its data. It is important that the communication staged through the host makes efficient use of both the PCIe and IB channels while the proxies cause minimal intervention to any application processes running on the host.

We discuss several approaches to design such a proxy, as part of the MVAPICH-PRISM framework. DirectIB provides a low latency path for inter-node communication from the MIC while host-based staging provides a high bandwidth path. It is critical for communication runtimes to provide a hybrid design that takes advantage of both options based on the communication characteristics.

Figure 5.3: Comparing Design Alternatives for Inter-node MIC-MIC Communication using IB
5.3.2 Direct IB

The OFA-CH3-IB interface in MVAPICH2 provides an implementation of MPI over IB verbs. Owing to CCL in MPSS, this interface can directly support MPI communication between processes running on a MIC and processes running on a remote Host or MIC. This path is depicted in Figure 5.3(a). An MPI transfer can be implemented as an IB transfer initiated by the mic-process. In the figure, we mark the initiator using a bolt symbol. Direct IB provides a low latency path for small message communication but the limitation in peer-to-peer bandwidth severely hinders the communication performance for large or concurrent messages. These limitations are expected to be more prominent when an MPI_Send is issued from a mic-process, as this involves a read by the IB adapter from the mic-process memory. In the following sections, we explore design alternatives to improve communication performance between remote MIC processes.

5.3.3 Passive Proxy

The first design that we consider for a proxy to handle host-staged communication from MIC aims at minimizing conflicts with any process running on the host. The proxy is only involved in setting up staging buffers on the host and is not directly involved in any communication. We achieve this by taking advantage of the RMA capabilities offered by SCIF and IB. There is an instance of the passive proxy launched on host of each node during the MPI job launch. During MPI library initialization, mic-processes establish a SCIF connection and an IB connection to the proxy running on the local host and requests buffer allocation. The proxy services an incoming request by allocating the requested buffer regions and registering it with the SCIF end-point of the corresponding process and the IB
protection domain. SCIF registration returns an offset that can be used by the remote process to access the buffer using RMA functions. IB registration returns an rkey that, coupled with the virtual address of the buffer, can be used in a similar fashion. This information is returned to the requesting process. The mic-process manages this buffer region as a pool of buffers that it uses for inter-node communication, as described later. Processes running on a MIC can co-ordinate to allocate a common buffer region at the proxy. The buffer region can be managed as a pool of buffers and shared between mic-processes through shared memory on the MIC. It is to be noted that SCIF registration information is specific to a process end-point and hence registration requests have to be made by and serviced for each mic-process individually, although they all share the same buffer region. Every process establishes an IB connection with the passive proxies running on all remote nodes. This can be done in an on-demand fashion as has been shown in earlier work [95]. The proxy goes to sleep once it services the initial requests of all the processes and until a mic-process signals it for extended buffer allocation or for MPI library finalization.

When a mic-process has to send a message to a process on a remote MIC or host, it pulls a buffer from its local host buffer pool and writes the data into it, using an RMA scif_writeto. This takes advantage of the high bandwidth PCIe channel between the MIC and host. As the write completes, it sends that virtual address of the host buffer and the corresponding IB rkey to the remote process through Direct-IB. We note that the overheads of this step are minimal, considering that the amount of data transferred to the remote process is small and the operation is not bandwidth sensitive. The remote process can issue an IB RDMA operation to read data from the remote host. The remote process sends a finish message, when the read completes and the buffer is released back to the pool, at the sender process. A similar procedure is used even if the remote process is on the host. It is to be noted that
the passive proxy is not involved in the communication. It is important for the communication runtime to make simultaneous use of the PCIe and IB channels to achieve maximum performance. We achieve this by pipelining SCIF and IB operations.

Figure 5.3(b) depicts a MIC to MIC inter-node transfer through the passive proxy. The sender mic-process initiates the SCIF write while the remote mic-process initiates the IB RDMA read. To be able to understand the performance characteristics of this design, it is important to analyze the performance characteristics of the channels involved. The transfer along the IB channel, as shown in the Figure 5.3(b), involves data movement from the host to a remote MIC. Though this channel does not incur high overheads, its peak bandwidth is limited to 5.2 GB/s. On the other hand, the data movement from MIC to its host is issued using the SCIF channel and can reach a peak bandwidth of 6.9 GB/sec. The passive proxy-based inter-node communication between mic-processes is hence limited by the bandwidth offered by the IB channel. In Section 5.2, we have shown how the performance of SCIF transfers differ depending on whether they are initiated on the MIC or on the host. The transfers initiated from MIC yield lower performance for medium messages, that will impact the overall performance of this design. In order to alleviate these overheads, we explore alternate designs that utilize the DMA engines that are available on the host processors.

5.3.4 Active Proxy

With an increasing number of cores per node and with architectures which are geared toward having a spare core, it can be viable to dedicate a core for the communication proxy. Our active proxy design considers this scenario and utilizes a core on the host to progress
communication. Through this, we take advantage of the best communication channels possible for data staging through the host.

**One Hop:** The first variant for an active proxy is derived from the design for passive proxy presented above. However, the availability of a dedicated processor core allows the proxy to initiate and progress communication that is staged through the host. The active proxy also takes care of allocating and managing the staged buffers on the host. In this design, there are proxies launched on the host of each node, as is the case with the passive proxy. The active proxy, after establishing SCIF connections with all the local mic-processes, listens for any incoming request messages. Like in the case of passive proxy, all mic-processes establish IB connections to proxies on the remote hosts on an on-demand basis. Each request message received from local mic-processes has information of the source and destination buffers between which the data has to be transferred. The proxy takes care of reading data from the local mic-process using SCIF and then writing the data into the remote process buffers using IB RDMA write. These two transfers are pipelined to utilize the PCIe and IB channels concurrently. The major advantage of this design, compared to that of passive proxy, is that the SCIF transfers are initiated from the host, and hence deliver higher performance for medium messages. This is depicted in Figure 5.3(c). This comes at the cost of sparing a dedicated core for communication progress. On the other hand, the overall performance is still bound by the bandwidth offered by the IB channel, as is the case with passive proxy.

**Two Hop:** The two-hop design for active proxy aims at taking advantage of the high bandwidth channels available between MIC and the host, on the same compute node. For a transfer between a mic-process and a remote mic-process, the data is first staged to the local host then to the remote host and finally to the remote mic-process. This flow is
depicted in Figure 5.3(d). Through this, it takes advantage of the SCIF channels and the high performing host-to-host IB channel. However, it pays the price of two hops that can result in added latency and pipelining overheads. The transfers are pipelined to minimize the impact of the additional hop. The functionality of this proxy is similar to that of the 1-hop case, except that message forwarding information has to be stored at the proxies.

5.3.5 Discussion

Each of the designs presented above have strengths and weaknesses. DirectIB provides a low latency channel while proxy-based designs aim at achieving high bandwidth. Passive proxy requires very little involvement from host cores, but suffers from MIC initiated SCIF transfers. The 1-hop and 2-hop active proxy designs take advantage of a dedicated core and try to use the best communication channels available. 2-hop chooses better channels compared to 1-hop but at the cost of an additional stage in the transfer pipeline. Active proxy can also become a bottleneck for request processing, as the number of processes on the MIC increases. Passive proxy allows for more parallelism in data movement, as each process issues and handles its own data movement operations. Choosing the best design to use depends on the user and the application. We offer the user a choice by selecting the design as a runtime parameter. If the user does not specify an option, then the selected design depends on the application characteristics. If an application uses all the compute cores in a node, we recommend the passive design in order to avoid conflicts with the existing host processes. Otherwise, we select the active design that aims to achieve peak bandwidth performance.
5.4 Experimental Evaluation

We evaluate the proposed designs for internode and intranode communication on different Xeon Phi clusters using different benchmarks, application kernels and applications. We present details of the experimental setup and performance results in the following subsections.

5.4.1 Intra-node Communication

Our experimental environment is a dual socket node containing Intel Sandy Bridge (E5-2680) dual octa-core processors, running at 2.70GHz, with 32GB of memory, a SE10P (B0-KNC) coprocessor and a Mellanox IB FDR MT4099 HCA. The host processors are running CentOS release 6.3 (Final), with kernel version 2.6.32-279.el6.x86_64. The KNC runs MPSS 4346-16 (Gold). We have used MVAPICH2 1.9a2 for our designs and experiments. The Intel compiler composer xe_2013.0.079 has been used. We have used OSU Micro Benchmarks (OMB) for all the micro-benchmark level evaluations.

In this section, we first present experiments studying the impact of our designs for Intra-MIC and MIC-host communication on performance of point-to-point operations. For intra-MIC communication, ‘SHM(DEFAULT)’ is the default version of MVAPICH2 1.9a2 using its shared memory channel. ‘SHM-TUNED’ is the same version with shared memory copy chunk size tuned to the DMA transfer length, as discussed in Section 5.1. ‘SCIF’ represents the design using the low level SCIF API for intra-MIC communication. For MIC-host communication, ‘IB(DEFAULT)’ is MVAPICH2 1.9a2 using its OFA-IB-CH3 interface and the native IB channel. ‘IB-SCIF’ is the same version of the code using the IB-SCIF channel. ‘SCIF’ is MVAPICH2 1.9a2 enhanced with SCIF-CH3 interface and uses symmetric model of DMA transfers depicted in Figure 5.2(b). ‘SCIF-HOST-INITIATED’
is similar to ‘SCIF’ except for it uses the host-initiated model of DMA transfers. In the later part of this section, we present performance evaluation for MPI Collective communication. ‘DEFAULT’ is the default version of MVAPICH2 using the default shared memory and IB channels. ‘SCIF-ENHANCED’ uses a hybrid selection of ‘SCIF’ and ‘SCIF-HOST-INITIATED’ designs based on the message size and collective operations as discussed in Section 5.2. Finally, we compare the performance of these two versions using a 3D Stencil communication kernel and P3DFFT application.

**Intra-MIC Communication:** As explained in Section 5.1, SHM channel based data transfers beyond a message size are orchestrated by transferring fixed sized chunks through pre-allocated shared memory buffers. Depending on the copy size, either the DMA or the processor cores are involved in the transfer. In the SHM-TUNED version, we adjust the copy chunk size to the DMA length of 64KB, thus taking better advantage of DMA-assisted transfers. SHM-TUNED improves the Intra-MIC MPI latency by up to 75%, for 4MByte messages, compared to SHM. It delivers a 4x improvement in both peak bandwidth and peak bi-directional bandwidth. The SCIF version optimizes this further by taking advantage of the lower level API to invoke the DMA and avoiding the additional copies by using direct memory access operations. SCIF delivers a 55% improvement in latency, compared to SHM-TUNED. It improves the bandwidth and bi-bandwidth by more than 3x and 5x, respectively. The results are depicted in Figure 5.4.

**MIC-Host Communication:** In this section, we discuss the results obtained with point-to-point experiments running one process on host and another on the MIC. From Figure 5.5, it is evident that IB-SCIF is not suitable for small sized messages due to the overheads of IB implementation over SCIF. The lines indicating the performance of IB-SCIF and SCIF designs depict this overhead. All the other designs perform similarly as they all
Figure 5.4: Performance of Intra-MIC Communication
Figure 5.5: Performance of Communication between MIC and Host
use native IB verbs underneath. IB(DEFAULT) however, relies on the HCA to make data transfers between the host and the MIC. This hurts the latency from medium to large sized messages. SCIF-based designs allow processes on the host and the MIC to communicate directly over the PCIe and hence have lower latency than IB for large messages. For 4 MB messages, we see close to a 70% improvement in latency with SCIF when compared to that using IB(DEFAULT). In Figure 5.5(b), the cross-over between IB-SCIF and IB(DEFAULT) arises from the counter-acting of software overhead involved with IB-SCIF and the gains obtained through DMA assisted SCIF-copies. Further, the difference in latency between the SCIF designs results from the non-symmetry for SCIF-HOST-INITIATED design, where all the sends are eventually dealt with by the DMA on the host processor, thus providing better latency. In the case of default SCIF design the average latency obtained is the result of transfers handled by both the DMA on the host and the proxy DMA on the MIC. For a 4 MB message, SCIF-HOST-INITIATED shows a 25% improvement over default SCIF design.

Figure 5.6: Uni-directional Bandwidth between MIC and Host Communication
Figure 5.7: Performance of Collective Operations with 8 Processes on Host + 8 Processes on MIC
An interesting result shown in Figure 5.5(c) is the degradation of bi-directional bandwidth for the SCIF-HOST-INITIATED design when compared to SCIF. This is because of the fact that SCIF-HOST-INITIATED uses the host DMA for both host to MIC and MIC to host transfers whereas SCIF uses both the host DMA in tandem with the proxy DMA on the MIC. This allows for better utilization of resources with SCIF. On the other hand, the results in Figure 5.6 demonstrate the benefits on uni-directional bandwidth of using the host DMA instead of the proxy DMA available on the MIC for MIC to host transfers. SCIF-HOST-INITIATED again shows 23% improvement for a 4 MB message over default SCIF design.

**Collective Communication:** Figure 5.7 shows the latency of some collectives with 8 processes on the host and the MIC each. Collective operations are built on top of point-to-point calls and hence directly benefit from the proposed designs. While DEFAULT design uses a combination of shared memory and IB semantics for data transport, the SCIF-ENHANCED design exploits SCIF API for large message transfers coupled with host-initiated or symmetric DMA selection as discussed in Section 5.2. SCIF-based designs improve the performance of 16 process Gather, Alltoall and Allgather collective operations by 70%, 85% and 80%, respectively, for 4MB messages, compared to the DEFAULT version.

**Application Level Evaluation:** We show the impact of our designs using a 3D Stencil communication kernel [86] with processes running across the MIC and the host. Each process in the process grid exchanges 64KByte messages with its neighbors in 6 dimensions. The processes are fit into as cubic a 3D grid as possible. The results are presented in Figure 5.8(a). We run the test with increasing number of processes on the MIC and host and we
see that the ‘SCIF-ENHANCED’ version consistently delivers benefits compared to ‘DEFAULT’. With 16 processes (8 on the host and 8 on the MIC), we see a 18% improvement in overall execution time per timestep.

Figures 5.8(b) and 5.8(c) show the impact on our design on P3DFFT, a popular library for parallel three-dimensional Fast Fourier Transforms [70]. It is written using Fortran and MPI. The version considered for our runs initializes a 3D array with a 3D sine wave, performs 3D forward Fourier transform and then backward transform in each iteration. It runs for 512 iterations and calculates an average execution time per iteration. We have tested it using two problem sizes and using 8 and 16 processes across the host and MIC. ‘SCIF-ENHANCED’ provides up to 11.5% improvement in performance compared to the ‘DEFAULT’ version of MVAPICH2. We see that, as the number of processes increases, the benefits from our enhanced communication runtime increases.

### 5.4.2 Inter-node Communication

We use the Stampede supercomputing system at TACC [15] for the experiments presented in this section. Each Stampede node is a dual socket containing Intel Sandy Bridge (E5-2680) dual octa-core processors, running at 2.70GHz. It has 32GB of memory, a SE10P (B0-KNC) coprocessor and a Mellanox IB FDR MT4099 HCA. The host processors are running CentOS release 6.3 (Final), with kernel version 2.6.32-279.el6.x86_64. The KNC runs MPSS 2.1.4346-16. The compiler suite used is the Intel Composer_xe_2013.2.146. We compare the performance of internode communication using the default OFA-CH3-IB channel in MVAPICH2 that uses DirectIB with the different proxy-based designs presented in Section 5.3. We use OSU micro-benchmarks 4.0 for
Figure 5.8: Performance of 3D Stencil Communication Kernel and P3DFFT Application
evaluation performance of point-to-point and collective communication. We then present evaluation using a 3DStencil communication kernel and the P3DFFT library.

**Point-to-point Benchmarks:** We present a comparison of the inter-node latency between processes running on MIC coprocessors in Figure 5.9. All the designs use DirectIB for transferring small size messages, as this offers the least latency path. As we move to larger messages, where the peer-to-peer bandwidth bottleneck comes into play, proxy-based designs outperform DirectIB. The Passive-Proxy design improves the latency for 4MByte messages by 57.5% when compared to the DirectIB-based design. The Active-Proxy-1Hop delivers better latency than the Passive-Proxy from the virtue of host initiated SCIF transfers [17]. For a medium length message, the Active-Proxy-2Hop looses out to Active-Proxy-1Hop by a small margin because of the extra hop involved. For large messages, it gains slightly from the high performance channels it uses. It delivers 65% improvement in latency compared to the DirectIB-based transfers. All the three proxy-based designs use pipelining to overlap SCIF and IB transfers.

![Graphs](image.png)

(a) Latency - Small Messages  
(b) Latency - Large Messages

Figure 5.9: Latency performance of inter-node MIC-MIC communication
Figure 5.10(a) shows a comparison of internode uni-directional MPI bandwidth using the different designs discussed in the report. The default version of MVAPICH2 that uses DirectIB is limited by the peak intra-IOH peer-to-peer PCIe read bandwidth on Sandy-Bridge platform which is around 1,000 Million Bytes/sec. As discussed in Section 5.3.3, the Passive-Proxy design achieves close to the peak IB write bandwidth from host to the remote MIC which is equal to the peak peer-to-peer write bandwidth from IB HCA to MIC. It provides a 4.6X improvement over the bandwidth offered by Direct IB, for 4MB messages. However, we observe that the bandwidth performance grows slowly with the message size which is because of the limited performance of MIC initiated DMA transfers. Active-Proxy-1Hop design overcomes the limitation in Passive-Proxy as the SCIF transfers are initiated by the proxy from the host. However, its bandwidth is also limited by peer-to-peer PCIe write bandwidth. Active-Proxy-2Hop tries in order to take advantage of the best channels available and overcome the limitations of peer-peer PCIe communication. It achieves the maximum uni-directional bandwidth performance among all the designs but does not achieve close to the peak bandwidth due to overheads in the two stage pipeline involved. The Active-Proxy-2Hop delivers up to 5.08x improvement in uni-directional bandwidth compared to DirectIB. Figure 5.10(b) shows the inter-node bi-directional bandwidth achieved using the different designs. Passive-Proxy delivers up to 2.75X improvement compared to the DirectIB design. However, we see that this is much lower than the peak bi-directional bandwidth that the IB channel between host and a remote MIC coprocessor offers. Our analysis shows that this is because of the rate at which completions are generated by the MPSS stack for the MIC initiated SCIF writes. When a large sequence of SCIF writes are issued, we have observed alternating phases of polling and back-to-back completion of multiple transfers. This results in idle phases for one of the channels
in the pipeline, thus limiting the bi-directional bandwidth achieved. Active-Proxy-1Hop, which uses host initiated SCIF transfers, avoids the issues and allows for a more balanced pipeline. It achieves up to 4.34X improvement compared to DirectIB. Active-Proxy-2Hop incurs considerable overhead due to the two hops involved. It delivers up to 3.54X improvement compared to DirectIB but loses to Active-Proxy-1Hop.

![Bandwidth and Bi-directional Bandwidth](image)

(a) Bandwidth  
(b) Bi-directional Bandwidth

**Figure 5.10: Bandwidth performance of inter-node MIC-MIC communication**

Figures 5.11 and 5.12 show the impact of our designs on the latency and bandwidth performance of communication between a MIC and a remote host. Notably, Active-Proxy-1Hop and Active-Proxy-2Hop are the same for MIC to remote host transfers as only the proxy at the node of the sending process is involved in the transfer. They perform better than Passive-Proxy as was the case with MIC to MIC transfers. They deliver close to a 55%, 4.76X, and 4.58X improvement in latency, bandwidth, and bi-bandwidth, respectively, when compared to DirectIB.

113
Figure 5.11: Latency performance of inter-node MIC-Host communication

Figure 5.12: Bandwidth performance of inter-node MIC-Host communication
Figure 5.13: Performance of Alltoall communication with 16 MIC coprocessors, up to 256 processes
Figure 5.14: Performance of MPI_Gather communication with 32 MIC coprocessors, up to 512 processes
Figure 5.15: Performance of MPI_Allgather communication with 32 MIC coprocessors, up to 256 processes
Figure 5.16: Performance of 3DStencil Communication Kernel with 32 MIC coprocessors, up to 512 processes
Figure 5.17: Performance of 3DStencil Communication Kernel with 32 MIC coprocessors and hosts, up to 1,024 processes
Figure 5.18: Performance of P3DFFT in Pure MIC configuration
**Collective Benchmarks:** Figure 5.13 shows the impact of the proposed designs on the performance of MPI_Alltoall communication operation among processes running on the MIC. With four and eight processes per MIC, we see that Active-Proxy-1Hop delivers the best performance among all designs. However, we see a clear trend of Passive-Proxy performing better as the number of processes per MIC increases. In the designs with an active proxy, the proxy receives and handles any data movement into and out of the coprocessor. With the alltoall communication pattern, this processing overhead increases significantly as the number of processes per MIC increases. One way to address this is to enable multiple threads/processes to act as proxies on the host. On the other hand, in the passive proxy mode, the communication is progressed by the processes themselves and allows for an even distribution of the load. This can potentially allow for better scalability. Passive-Proxy delivers 60% and 65% improvement in latency for a 256KByte Alltoall communication with 128 and 256 processes, respectively.

We present the performance results for MPI_Gather operation using the proposed designs in Figure 5.14. We see that active proxy-based designs benefit the performance while the Passive-Proxy does not provide any benefits. The MPI_Gather operation uses a binomial algorithm in this experiment. With a binomial tree, there are multiple processes sending to the co-root process of the sub-trees. In the case of the Passive-Proxy, the root and the co-root processes are required to issue and progress RDMA reads to receive data from their children. On the other hand, with the active proxy-based designs, the proxy of each child process writes into the memory at the co-root, thus providing parallel progress. This results in benefits using Active-Proxy-1Hop and Active-Proxy-2Hop. We see a 37% improvement in MPI_Gather operation with 512KByte transfer on 512 processes.
Figure 5.15 shows the impact of our designs on the MPI_Allgather operation with a varying number of processes running on MICs. With four processes/MIC and eight processes/MIC, we do not see any improvement in performance, using the proxy-based designs. To understand this behavior, we ran the experiments with only two processes/MIC and we see considerable impact of proxy based designs on performance. For large messages, MVAPICH2 uses the ring operation to implement the MPI_Allgather operation. The communication performance of the ring exchange is bound by the slowest communication path. As we increase the number of processes executing in the MIC, we observe that the latency of the intra-MIC exchanges increases and the intra-MIC transfers determine the latency of the Allgather operation. Hence, the true benefits offered by the proxy designs are not apparent with large message MPI_Allgather operations.

**Communication and Application Kernels:** We use a 3DStencil communication kernel to further evaluate the impact of the proposed designs. Nearest-neighbor communication pattern is very common in high-performance scientific applications. This communication pattern is latency critical and does not cause congestion in the network. Due to these characteristics, we see that any of the proxy-based designs shows a significant improvement in the communication performance. We see an improvement of 70% in the performance with 512KByte messages. We also run the 3DStencil benchmark with processes running both on the host and the MIC. When there are 16 processes running on each host, we observe the overhead of the active proxy’s intervention with the benchmark execution. Using Passive-proxy, we see an improvement of 56% in the performance with 1,024 processes on 32 nodes, for 512KByte messages. These results are shown in Figures 5.16 and 5.17.

Figure 5.18(a-d) shows the impact of our designs on P3DFFT, a popular library for parallel three-dimensional Fast Fourier Transforms [70]. It is written using Fortran and
MPI. The version considered for our runs initializes a 3D array with a 3D sine wave, then performs a 3D forward Fourier transform and then backward transform in each iteration. We run the kernel with two different data sets on 64 MICs, with 256 and 512 processes. We observe that, with 256 processes and 16 threads/process (4,096 threads), Active-Proxy-1Hop shows an improvement of around 19.9% and 20.7% in the overall execution time with 2Kx2Kx2K and 1Kx1Kx8K data grids, respectively. As process count increases, we see that Passive-Proxy shows increasing benefits. This confirms the behavior we have seen with MPI_Alltoall, the predominant communication in P3DFFT. With 512 processes and 16 threads/process (8,192 threads), we see up to 15.9% and 22.2% improvement in the overall execution time using Passive Proxy and Active-Proxy, respectively. We see an improvement of 27.5% and 42.7% improvement in the communication time, respectively.

**Analysis of PRISM:** We have seen that the different designs presented in the MVAPICH-PRISM framework deliver better performance with different communication patterns and scenarios. Figure 5.19 depicts this using a three dimensional space. Active proxy-based designs work around the channel limitations but incur staging and CPU utilization. This was evident in our experiments with MPI_Alltoall in Figure 5.13. As we increase the number of processes, the staging and CPU utilization overheads limit the performance benefits from better channel utilization in active proxy-based designs. Passive-proxy incurs no additional CPU utilization and addresses the key limitation in the inter-node communication path. For latency critical applications like 3DStencil, we have seen that it delivers close to the performance of the active proxy-based designs. It delivers higher performance than active proxy-based designs when a large number of transfers are concurrently staged through the host. In such cases, the active proxy becomes a processing bottleneck. All of the designs use DirectIB for small message transfers to take advantage of the low latency path.
5.5 Related Work

Researchers have explored various ways of programming applications to utilize MIC’s raw computing power. Being a fairly new terrain, most explorations have tried to make use of the architecture’s offload mode of computation. Koesterke et.al have shared early insights into the architecture’s data-parallel OpenMP-based programmability aspects [48]. Likewise, Deisher et.al have proposed a way to accelerate LU factorization by offloading computations on to the MIC [53]. MPI-based programming is a more natural fit for large scale applications such as the Weather and Research Forecasting (WRF) model. Larry Meadows’s work investigates the performance of WRF on the Xeon Phi using symmetric mode of computation [57]. MPICH2 1.5, an implementation of MPI, also has support for MIC architecture using shared memory, TCP/IP, and SCIF based communication [60]. Direct communication between MIC accelerators across nodes is supported by
Proxy-based designs to forward communication have been widely used in literature. Most of the frameworks for cluster-wide GPU virtualization [31, 96] use proxy-based designs to move data and to execute computation on remote GPUs. In addition, Intel MPI has a proxy-based design to overcome the PCI-e bottleneck on Sandy Bridge [12]. However, ours is the first work to discuss the design alternatives for proxy-based designs and analyze their tradeoffs. We enable the efficient use of MICs in the symmetric and many-core hosted modes at scale, on different system configurations. There has also been work from other researchers to model communication performance within a SMP Xeon Phi co-processor [33]. However, to truly harness the compute power of modern clusters with a large number of heterogeneous nodes, inter-node designs that allow efficient scaling becomes necessary.

5.6 Summary

In this chapter, we proposed designs for efficient MPI runtimes on clusters with Intel Xeon Phi co-processors, improving performance of Intra-MIC, Intra-Node and Inter-Node communication. Our designs take advantage of SCIF, Intel’s low-level communication API, in addition to standard communication channels like shared memory and IB verbs, to offer substantial performance gains in performance of the MVAPICH2 MPI library. We designed MVAPICH-PRISM: a proxy-based multichannel design in MVAPICH2 using IB and SCIF for Intel MICs. Our design allows applications to overcome the performance bottleneck imposed by state-of-the-art Sandy Bridge systems and extract the full compute potential of the MIC coprocessors. To the best of our knowledge, this is the first design that enables application developers to efficiently use the MIC in many-core hosted and symmetric modes with high performance and scalability. The proposed designs improve the point-to-point
latency from Xeon Phi to the host by 70%, for 4MByte messages, compared to an out-of-the-box version of MVAPICH2. Our solution delivers more than 6x improvement in peak uni-directional bandwidth from Xeon Phi to the Host and more than 3x improvement in bi-directional bandwidth. Using MVAPICH-PRISM, we improve inter-node latency between MICs by up to 65% and bandwidth by up to 5 times. MVAPICH-PRISM improves the performance of MPI_Alltoall operation by up to 65%, with 256 processes. It improves the performance of 3D Stencil communication kernel and P3DFFT library by 56% and 22% with 1024 and 512 processes, respectively.
Chapter 6: One-sided Semantics for Efficient Communication on Heterogeneous Clusters

Both MPI and PGAS models provide one-sided semantics that separate communication from synchronization. This separation allows application developers to design communication with minimal synchronization overheads and to maximize overlap between communication and computation. Earlier work has shown how one-sided communication can help irregular applications and applications which have bandwidth critical communication. In this chapter, we show the benefits of one-sided communication in a widely used seismic modeling code with regular communication pattern. We then demonstrate the performance benefits of using one-sided communication on heterogeneous clusters with RDMA like communication semantics offered by underlying hardware.

6.1 Optimizing Overlap in Applications: Case Study with a Seismic Modeling Code

6.1.1 Code structure of AWP-ODC

AWP-ODC is a community model [30, 62, 63] used by researchers at the Southern California Earthquake Center (SCEC) for wave propagation simulations, dynamic fault rupture studies, physics-based seismic hazard analysis and improvement of the structural models. The AWP-ODC code has been scaled to more than a hundred thousand processor
Figure 6.1: Data Transfer Patterns in AWP-ODC
cores, and consumed more than twenty million allocation hours in 2009 on the NSF Tera-Grid Ranger system at TACC, the Kraken XT5 at NICS, and the DOE INCITE Intrepid Blue Gene/P at ANL. The media has widely covered work using this code, including the TeraShake and SCEC ShakeOut-D exercises, and recent Wall-to-Wall simulations which are some of the most detailed simulations to date of earthquakes along the San Andreas fault in recent years. The ShakeOut-D simulations, for example, were part of the Great California ShakeOut Exercise which attracted ten million participants in 2009 to prepare for “The Big One.” The most demanding simulation of a wall-to-wall rupture scenario modeled an M8.0 earthquake with an 800x400x100-km3 domain at 100-m resolution for up to four minutes. It required 32 billion volume elements and 86,000 time steps, taking 20 hours on 32K Ranger processor cores, generating 7 terabytes of surface output and checkpoints. Recent work by Cui, et al. [28, 29], enhanced the application through single-processor optimizations, optimization of I/O handling and optimization of TeraShake initialization. In this work, we improve the application performance further by optimizing communication through latency hiding techniques.

AWP-ODC solves the 3D velocity-stress wave equation explicitly by a staggered-grid FD method, fourth-order accurate in space and 2nd-order accurate in time. The code includes a coarse-grained implementation of the memory variables for a constant-Q solid and Q relations validated against data. The code uses Perfectly Matched Layers to implement absorbing boundary conditions on the sides and bottom of the grid, and a zero-stress free surface boundary condition at the top. The 3D volume representing the ground area to be modeled is decomposed into 3D sub-grids. Each processor is responsible for performing stress and velocity calculations for its assigned sub-grid, as well as applying boundary conditions at any external edges of the volume contained within its sub-grid. Ghost cells,
**Main Loop in AWP-ODC**

```plaintext
do i = timestep0, timestepN
   Compute Velocities - $T_v$
   Swap Velocity data with neighboring sub-grids - $T_{cv}$
   Compute Stresses - $T_s$
   Swap Stress data with neighboring sub-grids - $T_{cs}$
endo
```

**Swap Velocity Data**

North and South Exchange
- `s2n(u1,north-mpirank, south-mpirank)`
  - `recv from south, send to north`
- `n2s(u1, south-mpirank, north-mpirank)`
  - `send to south, recv from north`
  - ... repeat for velocity components $v1,w1$

East and West Exchange
- `w2e(u1, west-mpirank, east-mpirank)`
  - `recv from west, send to east`
- `e2w(u1, east-mpirank, west-mpirank)`
  - `send to west, recv from east`
  - ... repeat for velocity components $v1,w1$

Up and Down Exchange
- ...

**S2N**
- Copy 2 planes of data from variable to send buffer
  - *copy north boundary excluding the ghost cells*
  - `MPI_Isend(sendbuffer, north-mpirank)`
  - `MPI_Irecv(recvbuffer, south-mpirank)`
  - `MPI_Waitall()`
  - Copy 2 planes of data from recvbuffer to variable
  - *copy into south ghost cells*

Figure 6.2: AWP-ODC Application Pseudo-Code
comprising a two-cell-thick padding layer, manage the most recently updated wavefield parameters exchanged from the edge of the neighboring sub-grids, see Figure 6.1 for a 2D schematic of these cells. The Ghost cell update presents a promising scenario to use one-sided communication for accelerated performance.

The main loop of the AWP-ODC is outlined in Figure 6.2. The velocity values are updated for the interior and the boundary of the volume. Then velocity values are exchanged with processors containing the neighboring sub-grids in the directions of north, south, east, west, up and down. This is followed by stress calculations and updates which are done in a similar manner. As the data grids are 3-dimensional, the boundary data along the non-major axes is a large number of small non-contiguous chunks. Therefore, intermediate staging buffers are used to accumulate and disseminate data at the source and destination, respectively.

6.1.2 Performance Analysis

In this section, we present an analysis of the AWP-ODC application execution time and present our case that improving the communication design can considerably improve the application run-time. As discussed above, the AWP-ODC application has a main loop where velocity and stress values are computed on each process and exchanged with its neighbors. As the complexity of the input data grid and the required accuracy increases, the number of iterations of the main loop increases. Figure 6.3(a) shows the time spent in the main loop as a percentage of the total application run-time on 4,096 processes. With 1,600 iterations, the main loop constitutes 96% of the total runtime. We used the mpiP profiling library [8] for this analysis. Real-world AWP-ODC runs generally involve several thousand main loop iterations with the largest ever involving 80,000 iterations. Thus, the
performance of the main loop code determines the overall application performance. In this
dissertation, we aim at improving the performance of the data exchange in the main loop
to improve overall application performance and scalability for large-scale runs.

**Communication Breakdown:** The main loop of the AWP-ODC involves computation
and exchange of velocity and stress. As shown in Figure 6.2, we refer to the time for
velocity computation as $T_v$ and velocity exchange/communication as $T_{cv}$. Similarly, the
time for stress computation is referred to as $T_s$ and the time for stress exchange is called
$T_{cs}$. The total run-time of the main loop in the case of ideal overlap would be: $\max(T_v, T_{cv})$
+ $\max(T_s, T_{cs})$. However, there are components in the communication times, such as the
data copies from and to the staging buffers, that cannot be overlapped in reality. A detailed
analysis of overlap with the actual times is presented in Figure 6.7.

Figure 6.3(b) shows that AWP-ODC spends 37% of its run time in MPI calls. The
MPI **Waitall** call, during which the communication progresses, contributes to 84% of the
MPI time. This progress of communication can be overlapped with computation using
RDMA capabilities of the underlying network thus improving the application performance.

### 6.1.3 Redesign using MPI Two-sided Non-blocking Semantics

The velocity and stress exchange routines account for most of the MPI time in the AWP-
ODC. Both velocity and stress have multiple components in them and each of these compo-
nents corresponds to a data grid. During the exchange, each process sends the boundaries
of these data grids to its neighbors in all directions and similarly receives boundary data
from them. As each data grid is 3-dimensional, the boundaries along the non-major axes
will be a large number of small non-contiguous chunks in memory. Each process accumu-
lates these chunks into contiguous staging buffers before sending them to its neighbors. A
corresponding dissemination happens at the receiving process.

In the original implementation, each of these exchanges is done in a blocking fashion
i.e. one exchange is initiated only after the previous exchange has completed. Such an
approach is required when there are data dependencies between the exchanges. We observe
that the application does not require such strict synchronization. Both the velocity and
stress computations are split in three and six independent components. Thus, there is an
opportunity for an asynchronous design that overlaps computation of components and their
communication.

![Figure 6.3: Analysis of AWP-ODC run-time](image)

(a) Percentage of application time spent in the main loop
(b) Percentage of application time spent in MPI calls

Figure 6.3: Analysis of AWP-ODC run-time
**Design enhancement #1:** In our first design, we use non-blocking two-sided calls to initiate exchange of all components of velocity and then wait on the completion of all transfers at once using a MPI_Waitall. The high level organization of code is shown in Figure 6.4. This allows for asynchronous progress. However, enabling concurrent transfers requires additional staging buffers as one staging buffer can be tied to only one transfer. This does not cause any additional overhead in the critical execution path as these buffers are created only once and exist throughout the application run.

**Design enhancement #2:** The computation of the different components in velocity and stress is also independent of one another. Leveraging this, we overlap the exchange of a component with the calculation of the next thus creating computation-communication overlap. To enable this, at a higher level, we split the velocity computation and swap functions into component level granularity and interleaved them as follows: Computation and exchange of velocity are split into three parts based on its three components $u$, $v$ and $w$. The value of $u$ is computed and the exchange of $u$ is overlapped with the computation of $v$. Similarly, the exchange of $v$ is overlapped with computation of $w$. For each component, we group the transfers to all neighbors to retain the advantage of concurrent progress explained above.

Though non-blocking two-sided semantics provide overlap with their underlying RDMA-based implementations, the skew created by the rendezvous handshake minimizes these benefits (since these are larger messages, typically of size 64K based on our problem size). We have explored the use of persistent send/recv in this context. The use of persistent requests can help slightly improve the latency of very small messages, they do not have any impact on medium and large message latencies. As they do not avoid the rendezvous handshake, their use does not improve the overlap characteristics. To overcome this limitation,
**Velocity Exchange**

\[ s2n(u1, \text{north-mpirank, south-mpirank}) \]

\[ ! \text{recv from south, send to north} \]

\[ n2s(u1, \text{south-mpirank, north-mpirank}) \]

\[ ! \text{send to south, recv from north} \]

\[ \ldots \text{repeat for east-west, up-down directions} \]

\[ \ldots \text{repeat for other velocity components} \]

\[ v1,w1 \]

\[ \text{wait \_onedirection()} \]

\[ s2nfill(u1, \text{recvbuffer, south-mpirank}) \]

\[ n2sfill(u1, \text{recvbuffer, north-mpirank}) \]

\[ \ldots \text{repeat for east-west, up-down directions} \]

\[ \ldots \text{repeat for other velocity components} \]

\[ v1,w1 \]

**S2N**

Copy 2 planes of data from variable to sendbuffer

\[ ! \text{copy north boundary excluding ghost cells} \]

\[ \text{MPI\_Isend(sendbuffer, north-mpirank)} \]

\[ \text{MPI\_Irecv(recvbuffer, south-mpirank)} \]

**WAIT\_ONEDIRECTION**

\[ \text{MPI\_Waitall(list of receive requests)} \]

**S2NFILL**

Copy 2 planes of data from recvbuffer to variable

\[ ! \text{copy to south ghost cells} \]

---

**Figure 6.4: Velocity Exchange with Reduced Synchronization using Non-blocking Two-sided Semantics**
we explored the use of one-sided semantics to further improve overlap. We describe our design using one-sided semantics in the next sub-section.

### 6.1.4 Redesign using MPI-2 One-sided Semantics

MPI-2 one-sided semantics are closely aligned with the network-level RDMA operations. Therefore, they are expected to provide better communication-computation overlap. MPI-2 provides three different kinds of synchronization operations in its one-sided model and choosing the one that suits the target application is crucial for good performance. As described earlier, AWP-ODC has a very localized communication, requiring each process to synchronize with only its nearest neighbors. A collective synchronization like that provided in MPI_Win_fence will lead to unnecessary overhead that increases with the number of processes. The Lock-Unlock semantics require processes to lock and unlock windows of each of its neighbors independently causing an overhead and reducing promise for concurrent communication progress.

We use the MVAPICH2 [9, 69] MPI library available on our target platform (TACC Ranger) for the experiments. MVAPICH2 implements several different modes and mechanisms for high-performance one-sided communication as described in [39, 45, 78]. However, at the time of our design and implementation, MVAPICH2 did not implement Lock-Unlock semantics using RDMA primitives. The lack of direct one-sided implementation of Lock-Unlock makes it less suitable for overlap. Thus, in this dissertation, we design our overlapping mechanisms using Post-Wait/Start-Complete semantics. The Post-Wait/Start-Complete model, with its group-based semantics, fits well with the nearest-neighbor communication pattern. The neighbors of each process form its Post and Start group.
MPI_Win_post(group, 0, window) ! pre-posting the window to all neighbors

**Main Loop in AWP-ODC**
- Compute velocity component u
- Start exchanging velocity component u
- Compute velocity component v
- Start exchanging velocity component v
- Compute velocity component w
- Start exchanging velocity component w
- Complete Exchanges of u, v and w
- MPI_Win_post(group, 0, window)
  
  ! For the next iteration

**Start Exchange**
- MPI_Win_start(group, 0, window)
- s2n(u1, north-mpirank, south-mpirank)
  ! recv from south, send to north
- n2s(u1, south-mpirank, north-mpirank)
  ! send to south, recv from north
  . . . repeat for east-west and up-down

**Complete Exchange**
- MPI_Win_wait(window)
- MPI_Win_complete(window)
- s2nfill(u1, window buffer, south-mpirank)
- n2sfill(u1, window buffer, north-mpirank)
  . . . repeat for east-west and up-down

**S2N**
- Copy 2 planes of data from variable to send-buffer
  ! copy north boundary excluding ghost cells
- MPI_Put(sendbuffer, north-mpirank)

**S2NFILL**
- Copy 2 planes of data from window buffer to variable
  ! copy into south ghost cells

Figure 6.5: Velocity Exchange with Reduced Synchronization using Post-Wait/Start-Complete
The receive staging buffers form the windows at each process. As window registration is an expensive collective operation, we remove this from the critical execution path by registering windows only once at the start of the application and retaining them until the end. With Post-Wait/Start-Complete semantics, the timing of posting the window for one-sided operations is crucial to achieve maximum performance and overlap. For example, if an MPI_Put is issued at a process before the window is posted by the target, the actual transfer is delayed until the first MPI call after the window is posted or even until the following synchronization where there are no further MPI_Put calls. This can reduce or totally negate the opportunity for overlap. This issue becomes more prevalent because of skew as the process count increases. To avoid this, in the AWP-ODC code, we post the windows before the main loop for the first iteration and at the end of each iteration for the following one.

The high-level organization of code is given in Figure 6.5. The computation and communication of different components in stress and velocity are overlapped. Transfers to all neighbors are grouped into one synchronization epoch for each component allowing concurrent communication progress.

### 6.1.5 Performance Benefits

We have run all of our experiments on the TACC Ranger system. Ranger is a blade-based system. Each node is a SunBlade x6420 running a 2.6.18.8 Linux kernel with four AMD Opteron Quad-Core 64-bit processors (16 cores in all) on a single board, as an SMP unit. Each node has 32 GB of memory. The nodes are connected with InfiniBand SDR adapters from Mellanox [5]. In our experiments, we used the MVAPICH2 1.4 MPI library [9, 69] installed on the Ranger. We use a weak scaling model for our experiments to
Figure 6.6: Comparison of various design enhancements (Left) main loop execution times (Right) percentage improvement

simulate the real world application use. We increase the size of the data set as the process count increases such that the data grid size per process remains at 64x64x64 elements. The message size, which is the size of the ghost cells in each direction, will be 64x64x2 real values i.e. 64K.

**Reduced Synchronization with Two-sided Non-blocking:** In this section we compare the performance of the original AWP-ODC application with the Async-2sided-basic and Async-2sided-advanced versions. Async-2sided-basic is the version described in Section 6.1.3 as design enhancement 1 and Async-2sided-advanced is the version redesigned for communication-computation overlap (design enhancement 2). Figure 6.6 (Left) shows the main-loop execution times of a 25 time-step application run for all the versions. Figure 6.6 (Right) shows the percentage improvements of the enhanced versions over the original version. In the performance analysis, we skip the first iteration from the timing to avoid
Timings from Original Main Loop

<table>
<thead>
<tr>
<th>$T_v$</th>
<th>$T_{cv}$</th>
<th>$T_s$</th>
<th>$T_{cs}$</th>
<th>Total Loop Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.40s</td>
<td>0.55s</td>
<td>2.45s</td>
<td>0.76s</td>
<td>4.16s</td>
</tr>
</tbody>
</table>

Theoretical Min Loop Time [100% overlap] ($\max(T_v, T_{cv}) + \max(T_s, T_{cs}) = 3.00s$)

Practically Non-overlappable Communication Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last velocity component ($T_{lv} = \frac{1}{3} T_{cv}$)</td>
<td>0.18s</td>
</tr>
<tr>
<td>Last stress component ($T_{ls} = \frac{1}{6} T_{cs}$)</td>
<td>0.13s</td>
</tr>
<tr>
<td>Intra-node communication ($\frac{T_{cv} + T_{cs}}{3} - \frac{T_{lv} + T_{ls}}{3}$)</td>
<td>0.34s</td>
</tr>
<tr>
<td>Copy costs for non-contiguous communication (measured)</td>
<td>0.11s</td>
</tr>
<tr>
<td>Total non-overlappable part</td>
<td>0.76s</td>
</tr>
</tbody>
</table>

Expected Main Loop Time Overlapping all Network Communication: $3.00s + 0.76s = 3.76s$

Measured Main Loop Time: $3.75s$

Figure 6.7: Component Analysis of the Original Main Loop Time

the impact of startup costs. We see an 8% improvement with the Async-2sided-basic version over the original version on 4,096 processor cores. With the Async-2sided-advanced version, we see an improvement of 11% on 4,096 processors cores and 6% improvement on 8,192 cores.

**Maximized Overlap using MPI2 One-sided Primitives:** We refer to our design of AWP-ODC based on the MPI-2 one-sided semantics as Async-1sided (Section 6.1.4). The high-level design of overlap is similar to the Async-2sided-advanced version; however, we see better performance because of the closer alignment of one-sided semantics with the underlying network RDMA operations. The advantage of overlap in Isend/Irecv model is hampered by the rendezvous handshake for large messages. With the async-1sided version we see up to 12% improvement over the original version on 4,096 processor cores and 10% improvement on 8,192 cores. The results are charted in Figure 6.6.

**Analysis of Achieved Overlap:** In this section we analyze the run-time of AWP-ODC in depth to better understand what percentage of application time was actually overlapped
using our designs. Velocity computation ($T_v$), velocity exchange ($T_{cv}$), stress computation ($T_s$) and stress exchange ($T_{cs}$) form the major part of the main loop in AWP-ODC. In case of ideal overlap, the total cost of the main loop would be $\max(T_v, T_{cv}) + \max(T_s, T_{cs})$. As shown in the table below, this time for a 25 iteration, 4,096 process run should be 3.0s. There are parts in $T_{cv}$ and $T_{cs}$ that cannot be overlapped for practical reasons. First, the copy of data from and to the staging buffers involves the CPU and cannot be overlapped. This copy time was measured as 0.11s during this run. Also, due to data-dependencies in the application, the communication of the last components in velocity and stress cannot be overlapped. We need to synchronize by calling MPI_Win_complete for the transfers to complete. This is due to data-dependency in the application. As velocity has 3 components and stress has 6 components, these times in velocity and stress would be $\frac{1}{3}T_{cv}$ and $\frac{1}{6}T_{cs}$, respectively. We performed our experiments on the TACC Ranger system, used for many of the very large real world AWP-ODC runs. Because of the limitations of setup on the cluster, intra-node transfers cannot use IOAT [89] or similar techniques and hence do not provide an opportunity for computation-communication overlap. In the 4,096 (16x16x16) process grid with 16 cores per node on Ranger, each process has its neighbors along the z-axis within the same node. This results in $(T_{cv} + T_{cs})/3$ being intra-node communication. Accounting for these components, the expected main loop time computes to 3.76s. Our actual runtime for the main loop, 3.75s, matches this expected value, getting 100% of the achievable overlap.
6.2 One-sided Communication on Heterogeneous Clusters

In this section, we demonstrate the benefits of using one-sided communication on heterogeneous clusters where underlying hardware offers RDMA-like communication semantics.

6.2.1 MPI-3 RMA using GPUDirect RDMA

In earlier chapters, we presented designs for MPI two-sided communication using GDR technology. On GPU clusters, buffering of messages becomes expensive when data is in GPU memory. This has lead MPI libraries to use rendezvous protocols, which ensure synchronization between the processes before data is transferred directly from source buffer to the destination buffer [50, 65]. This can add a significant overhead for small data transfers, limiting the benefits of GDR exposed to the application. Table 6.1 shows the overhead incurred by an MPI Send/Recv operation by comparing it with network-level latency.

<table>
<thead>
<tr>
<th>4 Bytes</th>
<th>Host-Host</th>
<th>GPU-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>IB Send/Recv</td>
<td>0.98</td>
<td>1.84</td>
</tr>
<tr>
<td>MPI Send/Recv</td>
<td>1.25</td>
<td>6.95</td>
</tr>
</tbody>
</table>

Table 6.1: Latencies (half round trip) of IB Send/Recv and MPI Send/Recv latencies for data movement between Hosts and GPUs (in usec)

The MPI Remote Memory Access (RMA) model allows developers to reduce the synchronization overhead per data transfer by separating data movement and synchronization into separate operations. Once a memory region is exposed, multiple data movement operations (Put, Get, Accumulate, etc.) can be issued before a synchronization call is made, to complete these operations. This amortizes the cost of synchronization. The MPI RMA
model also offers passive synchronization primitives which do not require any involvement from the target. The recent MPI-3 standard offers even more flexibility in managing data movement by separating the completion of data movement operations from synchronization operations. Recent studies have shown the impact of this extended model on CPU clusters [36]. The design of the MPI RMA model in general and the impact of MPI-3 extensions to the performance of communication has not been studied in the context of GPU clusters. The flexibility offered by the MPI RMA model also presents an opportunity to avoid the synchronization overheads incurred by traditional two-sided communication, as outlined above. This has the potential for significant benefits on state-of-the-art GPU clusters, compared to CPU clusters. However, PCIe bottlenecks in the current generation architectures limit the peer-to-peer bandwidth, which impacts the large message RMA performance, as discussed in Section 2.3.

In this chapter, we present designs for MPI RMA primitives on GPU clusters with GDR and analyze their potential for performance improvements compared to two-sided communication operations. We propose extensions to the model that can significantly reduce the synchronization overheads further and hence improve performance. We also propose proxy-based designs to efficiently support one-sided communication and overcome architectural bottlenecks. In the following sub-sections, we analyze the synchronization overheads when using two-sided communication and various MPI RMA synchronization semantics. We propose extensions that can help reduce these overheads. We also present a proxy-based runtime framework to efficiently support RMA model on GPU clusters with GPUDirect RDMA.
6.2.1.1 Analysis of RMA Primitives

In this section, we analyze the semantics of two-sided and one-sided synchronization operations, how they are implemented in RDMA-enabled MPI libraries, and potential overheads. We consider a typical scenario of communication where three things are achieved: 1) sending data to a peer process (using a put), 2) detecting the completion of transfer, and 3) notifying the peer of data’s arrival.

Send-Recv: In the case of two-sided operations, all of these are achieved as part of MPI_Send and MPI_Recv (Isend, Irecv, and Wait in the case of non-blocking) calls as these require explicit involvement of both the processes. At the runtime level, for GPU-to-GPU transfers send/recv calls are implemented using a rendezvous protocol for reasons explained in Section 3.2. An RDMA READ based implementation of this protocol involves three messages: request to send (RTS), data transfer, and finish message. The control messages add considerable overhead. The application level and runtime level activity during this transfer is depicted in Figure 6.8(a).

Post/Wait-Start/Complete: We now consider MPI one-sided communication using Post/Wait-Start/Complete (PW-SC) active synchronization. The communication calls themselves are non-blocking and do not ensure either completion or notification. Completion at the source is ensured by the MPI_Win_complete call while the call to MPI_Win_wait ensures that the data has arrived at the destination. This serves as the notification. RDMA-based implementations have to express synchronization as separate messages rather than piggy backing them with the data as in send/recv based implementations. This adds an overhead but is preferable as RDMA-based implementations deliver better overlap. Further, Complete-Wait calls mark the end of a communication epoch in the MPI RMA model.
The epoch should be reopened using Post-Start calls for communication. We include these in the communication time line shown in Figure 6.8(b).

**Flush:** The RMA model in MPI-3 provides more flexibility in the form of flush operations which ensure the completion of communication calls without requiring closure of the communication epoch. Flush calls can be made any number of times within one passive synchronization epoch. As depicted in Figure 6.8(c), MPI.Win.flush is a local operation and ensures only completion of communication. The developer has to use external synchronization mechanisms (eg. a barrier, send/recv, etc.) to notify the remote process that the data is available.

**Notify:** Another alternative MPI-3 provides for the developer is to use an atomic operation to set a flag in a remote processes window. The remote process can directly poll on the flag in the case of the “unified” memory model (public and private windows are the same) or poll with interspersed calls to MPI.Win.sync calls, in the case of “separate” memory model (public and private windows are separate, eg. non-cache coherent architectures). However, one thing to note is that the RMA model does not ensure ordering between these put and atomic operations. The user has to complete the put operation using a flush call and then issue the atomic operation, completing it using another flush call. This is depicted in Figure 6.8(d). The completion of individual messages incurs an overhead in direct RDMA-based implementations. These discussions would apply to host-to-host communication as well.

However, in this chapter, we focus on their impact on GPU-to-GPU communication, in the context of GPUDirect RDMA. Using an atomic operations and memory polling to achieve notification can be prohibitively expensive on GPU memory, as it would require a kernel launch or GPU to host copies. This can be avoided by using two windows, one
that exposes data memory on the GPU and one that exposes a notification region on the host. Another alternative is to use a dynamic window to which GPU and host buffers are attached. We will explore this further, in the following section. In later sections, we analyze the performance of these different alternatives and relate it to the discussion above. In the next section, we propose two simple extensions that have the potential of reducing the overheads mentioned above and hence to improve performance.

### 6.2.1.2 Extensions and Design

We first present the details of the extension, its API, usage and potential benefits. Later in this section, we present a design for the extension on InfiniBand.

**Extension:** We have analyzed how completion and notification is achieved using different synchronization operations in RMA and we outlined the overheads involved when implementing them over RDMA-enabled networks. In Figure 6.9(b), we have shown how atomic operations can be used to achieve notification rather than having to close an epoch or use an external two-sided based approached. However, the lack of ordering guarantees adds additional overhead as the data transfer has to be completed using flush before the atomic is issued. We propose an extension to the standard, a MPI_Put_notify call that helps address this overhead. Firstly, we note that widely used RDMA interconnects like InfiniBand guarantee the order between atomic operations and RDMA read/write operations issued before them. This can be exploited in this new call to avoid the explicit completion currently necessitated by the standard. Networks may also offer additional features, like RDMA write with immediate data in the case of InfiniBand, which can be used to optimize such an operation further. We present the implementation details later in this section. The parameters passed to the proposed call are as below.

```
MPI_Put_notify (void *origin_addr, int origin_count, MPI_Datatype origin_dt,
146```
The application provides location of the notification flag as a displacement into the window, `notify_disp`, which points to an 8-Byte region in the remote process’s memory. The notification buffer is set to the value “1” when the put completes. Memory polling on the notify flag is done similar to the way memory polling is done for detecting completion of atomic/accumulate operations in the current standard, as explained in Section 6.2.1.1. An efficient way to use this extension in the case of GPU-to-GPU communication is to have the notification buffer on the host. This can be achieved using a dynamic window to which both a GPU device buffer and a host notification buffer are attached. The displacement specified will be the values returned by the MPI_Get_address call as specified in the MPI 3.0 standard.

**Design of the proposed extension using IB primitives:** A direct implementation of the proposed MPI_Put_notify call over InfiniBand can be an RDMA write followed by an atomic operation as illustrated in Figure 6.9(a). InfiniBand ensures ordering of these operations and hence guarantees the semantics of the call. InfiniBand also offers an “IBV_WR_RDMA_WRITE_WITH_IMM” operation which piggybacks a 32bit value along with and RDMA write as shown in Figure 6.9(b). It causes a completion event on the remote process, which can be processed to get the immediate data. There are two key challenges in taking advantage of this operation to implement the extension. First challenge is the size of the data that can be piggybacked onto the RDMA write is limited to 32bits. This prevents us from sending the address of the notification buffer on 64bit machines which are ubiquitous. We address this by encoding the destination address as a combination of an index that identifies a window exposed on the remote process and the displacement into
the window. The number of bits used to represent these two parts is provided as a tunable parameter. This kind of an encoding enables us to take advantage of “WRITE_WITH_IM” operation in a larger set of scenarios. The second challenge is the progress required to process the immediate data at the target. In the “unified memory model” of the current MPI standard (3.0), updates to a remote process will eventually appear in its memory without the process having to make any MPI calls. On memory sub-systems which can support the “unified” model, the implementation using “WRITE_WITH_IM” operations can support the model with the help of a progress thread. On the other hand, the “separate memory model” requires the remote process to call MPI_Win_sync to synchronize public window to the private window before it observes the changes in its memory. This will ensure portability of MPI programs on non-cache coherent memory sub-systems or memory sub-systems where operations can be reordered. The “WRITE_WITH_IMM” based implementation can support this model without requiring a progress thread as the call to “MPI_Win_sync” can be used to progress the request and synchronize the flag from the runtime’s memory (public window) to the process’ memory (private window). In Section 6.3.1, we evaluate the performance impact of the proposed extension and the implementation on GPU-GPU communication.

6.2.1.3 Proxy-based Framework

GPUDirect RDMA allows the immediate application of different RDMA-based designs proposed in the context of CPU, for GPU clusters. However, the fact that GPU is connected as a PCIe device to the compute node presents additional challenges. Data transfers between InfiniBand adapter and the GPU are implemented as peer-to-peer transfers over the PCIe. State-of-the-art architectures pose bandwidth bottlenecks for movement of data along this path. In earlier chapters, we addressed these limitations in the context
of MPI two-sided communication using hybrid designs involving GPUDirect RDMA and host-based pipelining. Key additional features desired in the context of the RMA model are asynchronous progress and truly one-sided communication. One way to achieve this is to have a service thread per process that progresses communication. However, this will take additional CPU resources at each process and involves locking overheads as the service thread and the main process share the communication channel. In light of these requirements and challenges, we propose a proxy-based framework to support MPI RMA model on GPU clusters. The proxy uses CUDA memory copies and RDMA transfers to efficiently move data between GPUs, working around the bottlenecks. This extends our work to address similar issues in the context of Intel’s many-core co-processors.

Figure 6.2.1.3 depicts a high-level representation of the proxy-based framework. The key difference in the proposed framework from the proxy framework presented in earlier chapters is the way memory accesses are managed and how progress is made. When an RMA window is created (or a memory region is attached to a dynamic window), the proxy process running on the node will map the memory from all the processes running on the node into its address space using the CUDA Inter Process Communication (IPC) API. A challenge we face is that CUDA IPC is reliably supported between GPU devices only when they are connected on the same socket. We address this by managing multiple CUDA contexts for the proxy process. When there are multiple GPU devices per node, the proxy maintains a context on each GPU and keeps a mapping between MPI processes and the GPU device each of them uses. To avoid overheads of context switching, the proxy switches contexts only when mapping memory using CUDA IPC. We take advantage of the fact that access of device memory and check for progress of data movement in CUDA can
be done for all devices from any context. This is because of the virtue of unified virtual ad-
dressing provided by CUDA. When an RMA communication operation is issued, the MPI
process passes a signal to its proxy with information about the source and target buffers. As
the source buffer is not required to be inside a RMA window, this step may require setting
up additional IPC memory mappings, which are cached. The process is not involved in the
progress of the communication and is free until a synchronization call is made. The proxy
uses different communication paths depending on the location of the source buffer, desti-
nation buffer, and the architecture of the node (location of GPU and IB card), as presented
in our earlier chapters. For example, in Figure 6.2.1.3, the proxy is using a CUDA copy
from device to host and then an RDMA write to transfer data form host to the remote GPU.
This avoids the peer-to-peer read bottleneck present when GPU and IB card are connected
on the same socket. The proxy progresses these communication operations on behalf of all
processes on the node. As proxy is used only for large message communication, a single
proxy is enough to saturate the PCIe and network bandwidths. Small messages are handled
directly by the processes themselves.

### 6.2.2 MPI-3 RMA using CUDA IPC on Multi-GPU Clusters

In this section we present designs for MPI one-sided communication using CUDA IPC
which provides RDMA like communication between GPUs connected on a single node.
MPI one-sided communication semantics reduce synchronization overhead by separating
communication operations from synchronization operations. Processes can synchronize
once for a set of communication operations. On hardware with support for asynchronous
progress, these semantics can provide better performance through overlap compared to
send/receive semantics [73]. The CUDA IPC semantics coupled with the GPU DMA engine provides a good case for one-sided communication. The existing implementation of one-sided communication in MVAPICH2 is done on top of the send/receive operations. We describe our new designs to implement intra-node one-sided GPU-to-GPU communication below.

**Window Creation and Communication:** In the MPI one-sided model, each process exposes a region in its memory, a window, for direct access by other processes. When the window region is on the GPU device, the process creates a memory handle and exchanges with all the other processes within the node. Each process maps remote processes’ buffers into its own address space by opening their memory handles. Once this is done during the window creation step, Put and Get operations can be completed using memory copies to and from the IPC memory regions. For message sizes greater than 64KB, these operations are issued using asynchronous calls (cuMemcpyAsync) to enable overlap with computation operations on the CPU/GPU. For smaller messages, the overhead of explicit synchronization over-weighs any benefits of overlap. Accumulate operations can be implemented as CUDA kernels but are not the focus of this work.

**Synchronization:** All communication in MPI one-sided model is non-blocking. Their completion is ensured through synchronization operations. Synchronization can be Active, where both the origin (process issuing the communication operations) and the target (process whose window is accessed) are involved or Passive, where origin process can initiate and complete the communication operations by itself. Both these modes of synchronization for host-to-host communication are implemented using shared memory counters inside MVAPICH2 [68]. We extended these for GPU-to-GPU communication by using IPC events to order IPC communication before a synchronization call and operations that are issued
after it. Pair-wise IPC events are created between processes on a node during the window creation phase. We describe the use of events in different synchronization modes below.

MPI offers two kinds of Active Synchronization. `MPI_Win_fence` is a barrier like operation among all the processes in the communicator and any process can be an origin or a target. A Fence call starts the communication epoch and a second Fence call ends it. `MPI_Win_post`, `MPI_Win_wait`, `MPI_Win_start` and `MPI_Win_complete` provide a group based synchronization where each process can expose its window to a group of origin processes and can access windows at a group of target processes.

Post and Start calls start the communication epoch at target and origin processes, respectively. Wait and complete calls end the epoch at target and origin processes, respectively. In the synchronization calls that end a communication epoch, each origin process records IPC events created by each of its targets. The targets issue `cuStreamWaitEvent` on their events before exiting the synchronization call. This ensures that the communication to its window on GPU device memory completes before any future operation on this buffer is processed. The ordering between the record call at an origin and the wait call at a target is ensured by the host shared memory based synchronization mentioned in the context of CPU designs. All the record calls are issued before the host synchronization and wait calls are issued after the host synchronization.

In Passive synchronization, the origin process starts and completes a communication epoch at a target using `MPI_Win_lock` and `MPI_Win_unlock` calls, respectively. Locking can be exclusive or shared among processes and is also implemented using shared memory counters in MVAPICH2 [74]. In the modified design, when the target window buffer is on a GPU device, the origin issues a `cuStreamWaitEvent` call on the target’s IPC event immediately after acquiring a lock, to ensure that any earlier CUDA operation from the
earlier epoch is complete. It records the target’s IPC event before unlocking the window. The next process to acquire a lock on this window will wait on this event.

6.3 Experimental Evaluation

In this section, we present experiments that evaluate the performance and benefits of using one-sided semantics and the extensions we proposed, on GPU clusters. We used the Wilkes [92] cluster at University of Cambridge for these experiments. On Wilkes, each compute node is a dual socket machine with Xeon 6-core Intel IvyBridge processors operating at 2.6 GHz. Each node is equipped with 32 GB of memory, two NVIDIA K20c GPUs and two Mellanox FDR ConnectIB HCAs with PCI-Ex Gen3 interfaces. We used gcc GNU compiler 4.4.7 version, CUDA 5.5 NVIDIA toolkit. MVAPICH2.0b-GDR is used for experiments using two-sided communication based on GPUDirect RDMA [10]. All our enhancements and designs to MPI-3 RMA were implemented based on the open source version of MVAPICH2.0b library. We use OSU Micro-Benchmarks(OMB) suite for our micro-benchmark level evaluation. We extend it to measure the performance of RMA operations from GPU memory.

6.3.1 Performance using GPUDirect RDMA

We first evaluate the performance of MPI-3 RMA over GPUDirect RDMA using micro-benchmarks. We then use a couple of communication kernels which simulate the patterns commonly found in scientific applications, stencil exchange and all-to-all. We then study the impact of using one-sided semantics on non-contiguous data movement. Finally, we show the impact of our proxy-based framework on the performance for data movement with large message sizes.
**Micro-benchmarks:** Figure 6.11(a) compares the latency when using RMA communication operations for GPU-to-GPU communication with that of the latency when Send/Recv primitives are used. Put with active PSCW synchronization involves an RDMA write to signal start of the epoch, an RDMA write for the put and finally, another RDMA write for the completion. Despite involving three operations, it incurs less overhead than send/recv operation which involves two control messages and a round-trip for the RDMA read, as described in Section 6.2.1.1. Put with flush only ensures the completion of the RDMA operation issued to IB and does not include any notification to the target process. This significantly reduces the overhead of completing the Put operation.

Figure 6.11(b) shows the benefit of using one-sided transfers for GPU-to-GPU communication, in terms of bandwidth. As put operations directly translates to RDMA write on the network and synchronization between the processes happens only at the end of each window, 64 operations in the OSU benchmark for bandwidth. The bandwidth of two-sided operations is limited by the control messages involved in the rendezvous protocol. Figure 6.11(c) demonstrates the impact using message rate. With 12 pairs of processes running across two Ivy Bridge nodes, put operations backed by GPUDirect RDMA are able to achieve 11 Million operations/second between GPU memories. This is twice of what can be achieved using the Send/Recv implementation over GDR.

As discussed in Section 6.2.1.1, notification can be achieved without closing the communication epoch, using flush and then an external synchronization or an atomic operation. As shown in Figure 6.12(a), using the atomic operation incurs significant overhead, more than closing and reopening the active epoch. The proposed MPI_Put_notify extension enables us to avoid this overhead when backed by the RDMA write with immediate feature offered by IB. It takes only 2.78usec for half-a-roundtrip for inter-node transfers between
GPU accelerators, using GPUDirect RDMA. It provides a 60% improvement in latency compared to send/recv operations and more than a 40% improvement over existing one-sided primitives.

**Communication Kernels:** We use two very commonly found communication patterns to evaluate the impact of using one-sided primitives for data movement between GPU memories, when compared to traditional send/recv communication. Figures 6.12(b) and 6.12(c) show the performance of a stencil communication kernel that involves contiguous data transfers. This represents scenario where application packs its non-contiguous boundaries and uses MPI for data exchange. The stencil involves only one 3D data grid and hence requires synchronization at the end of each transfer. This incurs overhead in the case of Put with active synchronization implemented over RDMA, as explained in Section 6.2.1.1. Flush can be used to complete the communication but requires an external synchronization to signal other processes of the completion. This is achieved using an MPI_barrrier. Put with notify avoids the need for extra messages as the signal is piggybacked on the RDMA write in RDMA with immediate transfer. However, this involves synchronization at the remote process using MPI_Win_sync. We see that this does not incur an overhead in the case of stencil based communication pattern. It shows a 27% improvement in the execution time for the stencil exchange among GPU accelerators.

Figure 6.13 compares the different alternatives in the context of a dense alltoall communication exchange among GPUs. We see that, as we scale to more numbers of processes with dense communication pattern, the use of MPI_Put_notify backed by RDMA write with immediate data incurs an overhead. We attribute this to the overhead of processing the packets by target processes. Use of one-sided operations with flush and barrier as external
synchronization still achieves much better performance compared to send-receive based implementation. This again reflects the overhead incurred by send/receive operations in using GPUDirect RDMA for GPU-to-GPU communication.

Non-contiguous transfers are very common in applications and a good example is the boundary exchange involved in stencil-based computation. Application or the MPI library usually pack the data into a contiguous buffer before transferring over the network. When data is on the GPU, a CUDA kernel is used to efficiently pack/unpack data. This experiment compares the performance of using such a kernel to pack data before transferring over the network, with that of using GPUDirect RDMA to issue finer grained transfers over the network. Figure 6.14(a) shows the case of a stencil decomposed in 1D, where each plane is contiguous. This leads to transfers equal to the depth of the ghost cells, two in this example. In this case, using one-sided operations backed by GPUDirect RDMA provides better performance than packing the planes using a kernel. This pattern continues with a 2D decomposed kernel where each pencil is contiguous, as shown in Figure 6.14(b). However, as we move the 3D decomposition where each point on the boundary z planes is non-contiguous, using GPUDirect RDMA based one-sided transfer incurs an overhead due to the massive number of individual transfers involved. Packing data into a buffer get an advantage. It is to be seen how next-generation networks handle such fine-grained data movement.

**Impact of Proxy-based Framework:** Our proxy-based framework for RMA model helps alleviate the PCIe bandwidth bottlenecks described in Section 2.3. Figure 6.15(b) shows how it achieves 5.5 GB/sec for internode GPU-to-GPU communication compared to 3.0GB/s offered by PCIe P2P transfers on IVYBridge architecture. Figure 6.15(a) shows
the proxy’s impact on latency. As proxy manages communication progress, it achieves overlap with activity of the MPI processes.

6.3.2 Performance using CUDA IPC

Figures 6.16(a) and 6.16(b) compare the latency of Get with Lock-Unlock Synchronization. Compared to host shared memory based one-sided implementation (’1SC-SHARED-MEM’), IPC based implementation (’1SC-IPC’) improves the latency of 8Byte and 4MByte messages by 87% and 79%, respectively. ‘1SC-IPC’ shows 25% better latency than two-sided semantics implemented using IPC (’2SC-IPC’), for small messages. We see a similar trend for bandwidth. Figure 6.3.2 shows the passive progress that can be achieved by implementing one-sided communication over IPC. In this benchmark, the origin process issues 8 Get operations of 128KBytes each, using passive synchronization, while the target process in a busy loop for an increasing amount of time (using a timed while loop). We observe that the latency using ’SHARED-MEM’ increases as the busy loop at the target increases. On the other hand, ’IPC’ shows a constant latency due to the asynchronous progress offered by CUDA IPC.

6.4 Related Work

MPI-2 standard [58] expanded MPI to include Remote Memory Access (RMA) or one-sided communication. This mode of communication facilitates the coding of some applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing. There are proposals in the MPI forum to extend the current RMA interfaces [6]. Initial topics include exploiting different memory models, extending accumulate to support user-defined operations, extending accumulate to support read-modify-write operations, active message, etc. Our work targets at analyzing the potential of existing
one-sided and non-blocking two-sided semantics to exploit reduced synchronization and communication-computation overlap in the AWP-ODC application. Many researchers have investigated the techniques for communication/computation overlap. R. Brightwell et al. analyze the impact of communication and computation overlap [22], providing theoretical insights. Scientists further quantify this in large-scale applications [76]. An event-driven MPI library is designed to improve the communication responsiveness [55]. In [47, 88], communication progress is improved for zero-copy based design through interruption and helper thread respectively. MX Myri-10G [21] progression thread and the work [90] also have taken initial steps to improve Rendezvous progress by better utilizing the idle cores on a node within a multi-threaded infrastructure.

The Anelastic Wave Model (AWP) was originally developed by Kim Olsen et al. [30, 62, 63] and was picked as the primary model for the SCEC TeraShake simulation. As part of their work in [28, 29], Cui et al. enhanced the application through single-processor optimizations, optimization of I/O handling and optimization of TeraShake initialization. In this work, we improve the application performance further by optimizing communication through the aforementioned latency hiding techniques. The study in [59] explores the use of MPI one-sided semantics coupled with multi-threading to optimize the Community Atmosphere Model. Hermans et al. investigate the performance of one-sided communication alternatives in the NAS Parallel Benchmark BT application running on 256 cores of a Blue Gene/P [37].

Gerstenberger et.al have shown their work of implementing MPI-3 One-sided interface over RDMA networks with bufferless protocols [36]. Mirin et.al have published the work to improve the Community Atmosphere Model (CAM) by using MPI-2.2 One sided [19].
For the UPC programming model, Bell et.al have shown that one-sided communication model are more suitable for bandwidth-limited applications [20].

We distinguish this work from related efforts in being the first to combine GPUDirect RDMA with MPI-3 and proposed extended RMA features to improve performance.

6.5 Summary

In this chapter, we have shown the potential benefits of using one-sided communication semantics in MPI to overlap computation and communication in end applications. We first analyzed the communication in a widely used earthquake-induced ground wave-propagation simulation code, AWP-ODC, showing the potential for communication-computation overlap. Then we re-designed the application using MPI one-sided semantics to achieve the expected overlap. We see a 12% improvement in overall application performance on 4,096 cores. This effort was also part of the application’s entry as a Gordon Bell finalist at SC’2010.

Further, we demonstrated the potential performance benefits of using one-sided communication semantics on GPU clusters. We presented a novel design for MPI-3 RMA model on NVIDIA GPU clusters with GPUDirect RDMA. We performed detailed analysis of the potential impact of different synchronization primitives on communication performance, and proposed an extension to the MPI-3 RMA model that can reduce synchronization overheads. We also presented a proxy-based framework to efficiently support the MPI-3 RMA model on modern GPU clusters. We conducted an in-depth performance evaluation of different synchronization operations and extensions using micro benchmarks and communication kernels. The proposed extension to the RMA model enables an inter-node ping-pong latency of 2.78usec between GPUs, a 60% improvement over latency offered
by send/recv operations. One-sided communication provides 2x the message rate achieved by using MPI send/recv operations. The proposed extension improves the latency of a 3DStencil communication kernel, by up to 27%.
Figure 6.8: Communication Completion and Notification in Different Communication and Synchronization Mechanisms in MPI-3. (App: Application, RT: Runtime)
Figure 6.9: Implementation alternatives for the proposed extension

Figure 6.10: Proxy-based Framework to Efficiently Support RMA Model on GPU clusters
Figure 6.11: Comparison of Latency and Bandwidth Performance of Send/Recv and RMA Primitives
Figure 6.12: Performance of 3D Stencil Communication using Send/Recv and One-sided Primitives - Contiguous Data Transfers
Figure 6.13: Performance of Alltoall Communication using Send/Recv and One-sided Primitives

(a) 8 Processes

(b) 16 Processes

Figure 6.14: Performance of 3D Stencil Communication using Send/Recv and One-sided Primitives - Non-contiguous Data Transfers

(c) 3D Decomposition
Figure 6.15: Impact of Proxy-based Framework to Handle Large Message RMA Communication
Figure 6.16: Latency Performance and Passive Progress of Get with Passive Synchronization
Figure 6.17: Asynchronous Progress of Get with Passive Synchronization
Chapter 7: Impact on the HPC Community

In this Section, we highlight the impact this dissertation has on the HPC Community by influencing the way communication libraries are designed and through open-source releases.

7.1 Impact on the Design and Use of MPI Libraries on Heterogeneous Clusters

MVAPICH-GPU, proposed as part of this work, has changed the way MPI libraries are designed and used on heterogeneous clusters with GPUs. It introduced the concept of GPU-aware MPI communication to the HPC community. Many other MPI implementations like Open MPI [65] and Cray MPI [83] have followed suite to support MPI Communication from GPU memory and have employed optimization techniques similar to the ones proposed in this report. This work has also influenced the way MPI is used in applications being ported to GPU clusters. Application developers can now use MPI for communication directly from GPU device memory and let the MPI runtime optimize the data movement. This reduces the complexity of communication designs in applications.
7.2 Open Source Software Release and Wide Acceptance

MVAPICH2 [9, 69], is an open-source implementation of the MPI-2.2 specification over modern high-speed networks such as InfiniBand, 10GigE/iWARP and RDMA over Converged Ethernet (RoCE). MVAPICH2 delivers the best performance, scalability and fault tolerance for high-end computing systems and servers using InfiniBand, 10GigE/iWARP and RoCE networking technologies. This software is being used by more than 2,100 organizations world-wide in 72 countries and is powering some of the top supercomputing centers in the world, including the 7th ranked Stampede, 11th ranked Tsubame 2.5 and 16th ranked Pleiades (based on Nov 2013 Top500 Rankings). As of June ’12, more than 174,000 downloads have taken place from this project’s site. This software is also being distributed by many InfiniBand, 10GigE/iWARP and RoCE vendors in their software distributions. MVAPICH2-X software package provides support for hybrid MPI+PGAS (OpenSHMEM) programming models with unified communication runtime for emerging exascale systems. The duration of this work has spanned several release versions of the MVAPICH2 package, from version 1.5 to 1.9 (current). Most of the designs we have presented for efficient MPI communication on GPU clusters are already part of the MVAPICH2 1.8 and 1.9 software distribution. They are widely used on large scale GPU Clusters including Tsubame 2.0 and Keeneland. The designs for efficient MPI communication on MIC clusters have been incorporated into the MVAPICH2 code-base and are incrementally being deployed on NSF-TACC Stampede, the largest cluster with Intel MIC based co-processors that is available for open science. All components of this thesis will eventually be available as part of the MVAPICH2 and MVAPICH2-X packages.
Chapter 8: Future Research Directions

This chapter describes the possible future research directions that can be explored as a follow up of the work done as part of this thesis.

8.1 Efficient Streaming of Communication and Computation tasks on GPU clusters

The work done in this thesis has significantly improved the performance of communication from device memory on GPU clusters. The designs proposed in Sections 3 and 4 optimize data movement using techniques like CUDA IPC and GPUDirect RDMA, which can completely bypass CPU memory. The extensions we proposed to standard programming models like MPI and OpenSHMEM allow the runtimes to do this transparently, hiding the complexity of underlying system from the user. However, it is to be noted that the communication operations and computation kernels still have to be initiated and completed by the application process running on the host. The process also has to take care of ordering of communication expressed using an MPI/PGAS model and computation expressed using CUDA. This incurs an overhead at the start and end of the different computation and communication phases of the application. This overhead grows as applications scale to larger clusters. For example, launching and termination attribute to a considerable amount of the runtime in short running GPU kernels. A mechanism to offload the dependencies between
the communication and computation phases onto the GPU allow the scheduler to hide some of these overheads. Further, enabling communication from inside the GPU kernels completely removes the need of the synchronization before communication can be initiated. Achieving this will require novel designs at multiple layers in the system software stack, including InfiniBand/CUDA drivers as well as communication runtimes. It is essential that these features are exposed through standard programming models like MPI and OpenSHMEM. This will allow application developers to take advantage of the new features with minimal programming and performance overheads.

8.2 Runtimes for Highly Parallel Execution Environments

Intel MIC architecture, unlike GPU, allows application processes to run directly on the co-processor. Typically, threading is used to take complete advantage of the parallelism and compute power available on the Xeon Phi. However, even in state-of-the-art applications, communication is still issued and progressed by a single thread. This forces synchronization among the threads at the end of each computation phase and also does not take advantage of the massive parallelism available during communication phases. Lack of runtimes that can efficiently handle concurrent data movement operations in a multi-threaded environment is one of the main reason that forces developers to resort to the above mentioned approach. There has been some initial research done on designing multi-endpoint runtimes which can help alleviate this limitation on multi-core clusters [51]. There is also an effort underway in the MPI Forum that extends the MPI standard to efficiently support communication in mutli-threaded (MPI+OpenMP) applications. Addressing the above mentioned issues will be come even more critical as more and more applications are ported to the
massively parallel environments with Intel Xeon Phis. In Sections 5.1, 5.2 and 5.3, we presented several runtime level designs that improve performance of data movement of Xeon Phi clusters. Researchers can explore the application of the proposed designs in highly-parallel environments, taking multi-endpoint runtimes and MPI-level extensions into consideration. It is important that applications are redesigned using programming models and runtimes, as they evolve to support highly-concurrent communication regimens.

8.3 Impact of Designs on Energy-constrained Environments

Energy-constrained environments will become increasingly common as the HPC community heads towards exascale computing. Heterogeneous configurations with massively parallel accelerators and co-processors coupled with low-power host processors (e.g. ARM) are being explored as a possible alternative for building such systems. For applications to efficiently run on such clusters, computation kernels ought to be designed in a highly parallel fashion to take advantage of the accelerators or co-processors while the host processors are only involved in scheduling/progressing computation and communication phases. The work done as part of this thesis demonstrates how one-sided communication semantics backed by RDMA-based runtime-level designs can significantly reduce synchronization overheads and can completely hide communication overheads through overlap. The need for such designs will become imperative as we move into energy-constrained environments with slower host processors described above. One-sided, offloaded communication will reduce the overhead on the low-power processor preventing it from becoming the bottleneck in the system. As these architectures mature, research to evaluate the impact of these designs on energy-footprint of applications will be essential. New runtime-level
techniques and programming model-level extensions may have to be devised based on the requirements and capabilities of these new architectures.
Chapter 9: Conclusion and Contributions

GPUs are becoming a common component in modern system architectures due to the high compute per watt they offer. High performance networks, on the other hand, are helping these clusters to scale to thousands of nodes. Applications from a wide range of fields that use MPI and PGAS models are being ported by take advantage of the compute power offered by these heterogeneous clusters. They continue to use MPI or PGAS models to express parallelism across the cluster but have to use models like CUDA, OpenCL and others to express parallelism within a GPU accelerator. The requirement that two different models (and their runtimes) have to be used to move data on these clusters has rendered it complicated to optimize communication in applications. The increased complexity of the system architecture and the varying communication performance characteristics based on system configuration make it almost impossible for application developers to extract the maximum performance from the underlying communication subsystem.

In this dissertation, we have enabled MPI and OpenSHMEM PGAS models to support communication directly from GPU device memory. This considerably simplifies the communication code in applications running of GPU clusters. It also allows MPI and OpenSHMEM runtimes to optimize data movement while hiding the associated complexity from the user. We take advantage of Unified Virtual Address space offered by CUDA to enable use of standard MPI calls for communication from GPU memory, without any changes to
the interface. We propose simple extensions to the OpenSHMEM memory model to enable
developers to use OpenSHMEM for communication from GPU memory. We ensure that
the extended model is interoperable with both CUDA and OpenCL. Further, we proposed
designs in MPI and OpenSHMEM runtimes to optimize data movement across different
configurations of GPUs on clusters. We use techniques like pipelining and take advantage
of advanced features like IPC and GPUDirect RDMA to extract maximum performance of-
fered by the underlying hardware. The designs proposed for MPI are already incorporated
into the MVAPICH2 MPI library which is used on several large scale deployments of GPU
clusters with InfiniBand interconnect. We have shown up to 69% and 45% improvement
in point-to-point latency for data movement for 4Byte and 4MB messages, respectively.
Likewise, the solutions improve the bandwidth by 2x and 56% for 4KByte and 64 KByte
messages, respectively. Our work have been released as part of MVAPICH2 packages,
making it the first MPI library to support direct GPU-GPU communication. It is currently
deployed and used on several large GPU clusters across the world, including Tsubame 2.0
and Keeneland. The proposed extensions to OpenSHMEM, coupled with an efficient run-
time, improve the latency of 4 Byte \texttt{shmem\_getmem} latency by 90%, 40% and 17%, for
intra-IOH, inter-IOH and inter-node GPU configurations with CUDA, respectively. They
improve the performance of Stencil2D and BFS kernels by 65% and 12% on clusters of
192 and 96 GPUs respectively.

The Many Integrated Core architecture from Intel allows the same programming mod-
els to be used across both host and coprocessors. However, the complexity of system
configurations, the myriad channels provided by the communication subsystem and unique
performance characteristics of each of these channels considerably increase the complexity
of optimizing communication on these clusters. As part of this dissertation, we present
designs in MVAPICH2 MPI library to optimize communication on nodes with Xeon Phi coprocessors. We have demonstrated significant improvement in performance at micro-benchmark and application level, when compared to that of an out-of-the-box version of MVAPICH2. The proposed designs use a hybrid approach by taking advantage of the best performing channels based on the location of the processes and the communication pattern. We explored designs for internode communication on MIC clusters. The proxy-based framework efficiently tackles the challenges that state-of-the-art system architectures pose in achieving optimal communication performance. The proposed designs deliver up to a 70% improvement in the point-to-point latency and more than 6x improvement in peak uni-directional bandwidth from Xeon Phi to the Host. Using MVAPICH-PRISM, we improve inter-node latency between MICs by up to 65% and bandwidth by up to 5 times. MVAPICH-PRISM improves the performance of MPI_Alltoall operation by up to 65%, with 256 processes. It improves the performance of 3D Stencil communication kernel and P3DFFT library by 56% and 22% with 1024 and 512 processes, respectively.

Computation and communication overlap has been a critical requirement for applications to achieve peak performance on large scale systems. MPI and PGAS models offer one-sided communication primitives that separate communication from synchronization allowing for better overlap and reduced synchronization overheads. In this thesis, we demonstrate how even applications with regular communication like nearest-neighbor pattern can take advantage of one-sided primitives to optimize communication through communication-computation overlap. Further, we show how one-sided primitives can help achieve better intra-node communication performance than two-sided primitives on heterogeneous clusters with GPUs. We have shown a 12% improvement in overall application performance of a seismic modeling code, AWP-ODC, on 4,096 cores. This effort was also
part of the application’s entry as a Gordon Bell finalist at SC’2010. Our one-sided designs take advantage of RDMA like semantics offered by CUDA IPC and GPUDirect RDMA for communication between GPUs within a node and across a cluster, respectively. We also proposed extensions to the MPI-3 RMA model that can enable significantly better performance through reduced synchronization overheads during communication on GPU clusters. We have shown the benefits of one-sided communication operations using communication kernels and motivated the need of application developers to use one-sided communication semantics for redesigning applications for heterogeneous clusters. The proposed extension to the RMA model enables an inter-node ping-pong latency of 2.78usec between GPUs, a 60% improvement over latency offered by send/recv operations. One-sided communication provides 2x the message rate achieved by using MPI send/recv operations. The use of one-sided semantics improves the latency of a 3DStencil communication kernel, by up to 27%.
Bibliography


[9] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.

[10] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.


181


183


