High-Performance Heterogeneity/ Energy-Aware Communication for Multi-Petaflop HPC Systems

Dissertation

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

By

Akshay Venkatesh, M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2017

Dissertation Committee:

Dhabaleswar K. Panda, Advisor
Ponnuswamy Sadayappan
Radu Teodorescu
Abhinav Vishnu
Khaled Hamidouche
© Copyright by
Akshay Venkatesh
2017
Abstract

Increasing machine throughput by increasing frequency alone is limited by power constraints. This has forced high-performance computing (HPC) systems to evolve into complex machines with very high core counts and node counts, often, augmented with accelerators and co-processors with their own memory and network subsystems. Two distinguishing architectural traits of today’s multi-petaflop HPC clusters that marks a paradigm shift from previous generations are the presence of many-core processors, and high-speed network devices with specialized hardware features. A large proportion of machines on the current TOP500 list of supercomputers exhibits such characteristics. These changes have come about with a two-fold end goal of minimizing both execution time and energy expenditure of scientific applications that use these systems.

To reduce the time of compute phases, the high degrees of parallelism available on many-core processors are exploited. Of these processors, NVIDIA general-purpose graphics processing units (GPGPU) and Intel Many Integrated Core (MIC) processors are popular owing to their low power footprint and their ability to potentially produce over a teraflop/second throughput. The complexity with these processors lies in that they are often (if not always) available as PCIe devices with their own memory and network subsystems. This renders the nodes of the system heterogeneous from both a processor, memory, and the network perspective. To accommodate for this heterogeneity and to improve processor utilization in general, Network Interface Controllers (NICs) are being designed with a
range of novel capabilities to reduce communication time and increase overlap possibilities. These include the ability to access the memory of these PCIe devices even when they are located on remote nodes directly through Remote Direct Memory Access (RDMA) and multicast features. In addition, NICs are being designed with the capability of accepting a list of basic data movement and dependency satisfaction primitives that can be used to design communication routines that do not require CPU intervention for progression. Lastly, NICs and accelerators such as NVIDIA GPUs are being co-designed to allow the GPU to operate autonomously and issue network operations nearly independent of the CPU. This bears the potential to realize efficient control plane decoupling and achieve an overall increase in compute-resource utilization.

While reducing execution time is important, saving system energy is equally vital. One of the main contributors to the system’s energy consumption is the CPU as it is common to employ polling schemes for communication latency optimization. To reduce the energy footprint of processors, power-knobs such as Dynamic Voltage Frequency Scaling (DVFS), and interrupt-driven execution modes are often used. These are especially important during communication phases as they often result in energy expenditure far larger than that in compute-phases as recent studies have indicated.

In view of these developments, communication runtimes must aim to take into consideration the abundance of heterogeneity that abounds in modern HPC systems and leverage advancements in network design to achieve low latency, high throughput and to increase computation/communication overlap possibilities. At the same time, they must also aim to reduce energy expended by CPUs during communication phases with minimal or without affecting overall performance. As Message Passing Interface (MPI) is the de facto
standard for communication calls in scientific applications, this dissertation proposes high-performance designs while addressing heterogeneity and energy challenges for the MPI communication routines on modern HPC systems.

In particular, this dissertation attempts to address heterogeneity challenges in communication algorithms for dense collective operations through the design of novel heterogeneity-aware collective designs. The proposed designs are considerate of the cost differences in communication paths and leverage delegation mechanisms, propose adaptations to classic algorithms, and use novel heuristics to achieve large benefits in collectives (such as MPI_Alltoall and MPI_Allgather) at scale. The dissertation also takes advantage of special-capability network subsystems that have peer-to-peer access with other PCIe devices, RDMA and multicast capabilities, offload mechanisms, and the ability to service network requests without explicit CPU intervention. These are used to design point-to-point and collective operations that overcome the challenges posed by heterogeneous memory subsystems and yield high throughput, overlap, and reduced coupling-induced synchronization overheads. Finally, the dissertation proposes rules for application-oblivious energy savings during 2-sided and 1-sided MPI routines through intimate knowledge of the underlying protocols used to realize communication routines. This results in applications automatically saving CPU and memory energy during communication phases with minimal performance degradation and without needing application code changes. The proposed designs for MPI have been integrated into the MVAPICH2 MPI library and are already widely deployed on HPC clusters.
To my brother, my parents, mentors, and friends.
Acknowledgments

I thank my brother, Manasij, for constantly inspiring me in nearly all facets of life. I also thank my advisor, Dr. DK Panda, for the opportunity to work with him. His pragmatism, technical vision, and his ability to lead by example have been constant sources of motivation to strive towards academic excellence. I thank Dr. Sreeram Potluri and Dr. Khaled Hamidouche for serving as irreplaceable mentors during the course of my doctoral studies. Most members of the NOWLAB have contributed towards the research conducted as part of this dissertation in one way or the other. I thank all my lab mates over the years for this. I thank my professors for instilling a sense of enthusiasm during graduate studies. Lastly, I thank my parents for their dedication and love, and my close friends for their support, encouragement, and for comforting me with good vibes when most needed. This work is dedicated to all the above.
Vita

2007-2011 ........................................B.Tech., Information Technology, National Institute of Technology Karnataka, India

2011-Present ................................. Ph.D., Computer Science and Engineering, The Ohio State University, U.S.A

2012-Present ................................. Graduate Research Associate, The Ohio State University, U.S.A

2014 ................................................. Summer Research Intern, Pacific Northwest National Laboratory, U.S.A.

Publications


S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy and D. Panda, Efficient Inter-node MPI Communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs Int’l Conference on Parallel Processing (ICPP ’13), October 2013


Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvi</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Statement</td>
<td>8</td>
</tr>
<tr>
<td>1.2 Research Framework</td>
<td>10</td>
</tr>
<tr>
<td>2. Background</td>
<td>18</td>
</tr>
<tr>
<td>2.1 InfiniBand</td>
<td>18</td>
</tr>
<tr>
<td>2.1.1 IB Verbs and Available Transports</td>
<td>18</td>
</tr>
<tr>
<td>2.1.2 Remote Direct Memory Access (RDMA) using InfiniBand</td>
<td>19</td>
</tr>
<tr>
<td>2.1.3 Hardware Multicast Capability using InfiniBand</td>
<td>20</td>
</tr>
<tr>
<td>2.1.4 Scatter-Gather-Lists (SGLs) in InfiniBand</td>
<td>20</td>
</tr>
<tr>
<td>2.1.5 Offloading Network Operations in InfiniBand</td>
<td>21</td>
</tr>
<tr>
<td>2.2 Programming Models for High Performance Computing Scientific App-</td>
<td>21</td>
</tr>
<tr>
<td>lications</td>
<td>21</td>
</tr>
<tr>
<td>2.2.1 MPI</td>
<td>22</td>
</tr>
<tr>
<td>2.2.2 GPU Architecture and CUDA Programming Model</td>
<td>25</td>
</tr>
<tr>
<td>2.2.3 Intel MIC Architecture and Programming Model</td>
<td>29</td>
</tr>
<tr>
<td>2.3 Architectural Bottlenecks with PCIe Devices</td>
<td>32</td>
</tr>
</tbody>
</table>
2.4 Levers for Energy Conservation in Applications and Power Monitoring . . 34
   2.4.1 Power Levers .................................................. 34
   2.4.2 Power Measurement .......................................... 35

3. Heterogeneity-Aware Dense Collective Designs for InfiniBand MIC Clusters . 36
   3.1 Cost Model ............................................................ 36
   3.2 Heterogeneity-Aware Allgather Designs .......................... 38
      3.2.1 Short-Message Allgather Collective Designs .............. 38
      3.2.2 Large-Message Allgather Collective Designs .......... 42
   3.3 Heterogeneity-Aware Alltoall Designs ........................... 45
      3.3.1 Short-Message All-to-All Collective Designs .......... 45
      3.3.2 Large Message All-to-All ................................. 46
   3.4 Performance Evaluation ........................................... 50
      3.4.1 Micro-benchmark Evaluation ............................... 51
      3.4.2 Evaluation with an Application Kernel .................. 58
   3.5 Related Work ........................................................ 59
   3.6 Summary ............................................................. 61

4. Leveraging Novel Network Capabilities for High Performance GPU Collectives 62
   4.1 Challenges in Multicasting GPU Data and Design Alternatives .......... 62
      4.1.1 Using an Eager Protocol ..................................... 64
      4.1.2 Using a Rendezvous Protocol ............................... 65
      4.1.3 Proposed Design - GDR-SGL-MCAST ....................... 66
   4.2 Design Considerations for Dense GPU Non-blocking Collectives (NBCs)
      and Proposed Approaches ........................................ 70
      4.2.1 Progression Through Host-assistance ..................... 71
      4.2.2 Progression Through Host-assistance and Offload Mechanisms .. 71
      4.2.3 Progression Through Offload Mechanisms and GDR .......... 72
      4.2.4 Progression Through Offload mechanisms, GDR and CUDA Call-
      back Mechanisms ................................................. 73
      4.2.5 Generalization Through Schedule Abstraction Reuse .......... 74
   4.3 Performance Evaluation ............................................ 76
      4.3.1 Latency of multicast-based broadcast operation .......... 79
      4.3.2 Throughput of broadcast operation ....................... 82
      4.3.3 Latency of MPI broadcast .................................. 83
      4.3.4 Overlap performance of MPI_Ialltoall operation .......... 85
      4.3.5 Latency performance of MPI_Ialltoall operation .......... 86
      4.3.6 Overlap/Latency performance of MPI_Iscatter operation ... 86
      4.3.7 Overlap/Latency performance of MPI_Igather operation .... 87
<p>| 4.3.8 | Overlap/Latency performance of with more than 1 MPI process-per-node | 88 |
| 4.3.9 | On Latency comparison with blocking variants and on the effects of running GPU workloads | 88 |
| 4.4 | Related Work | 98 |
| 4.5 | Summary | 99 |
| 5.1 | Overheads From CPU-managed Synchronization Between CUDA Compute and MPI Phases | 102 |
| 5.2 | Mitigating Synchronization Costs Stream-based MPI: Extensions | 105 |
| 5.2.1 | General Vision For Control Flow Offload to GPU | 105 |
| 5.2.2 | Semantics of Issuing MPI Operations on Streams | 107 |
| 5.2.3 | Alternatives for Stream Association with MPI Operations: | 108 |
| 5.2.4 | Informal Definition of Semantics of MPI Operations on Streams: | 109 |
| 5.2.4.1 | Semantics of Send and Receive Operations on Stream: | 109 |
| 5.2.4.2 | Progressing Communication Calls Offloaded to the GPU: | 110 |
| 5.3 | Mitigating Synchronization Costs Stream-based MPI: GPU-initiated Communication Designs using GPUDirect-aSync (GDS) | 111 |
| 5.3.1 | Overview of CPU, GPU and HCA interaction under GDS | 111 |
| 5.3.2 | CUDA API Enhancements | 113 |
| 5.3.3 | OS-level Enhancements | 114 |
| 5.3.4 | IB Verbs Enhancements | 115 |
| 5.4 | MPI-level Designs for Decoupling CPU-GPU Control Flow | 116 |
| 5.4.1 | Protocols for Enqueuing MPI Point-to-point Communication Operations onto Streams | 116 |
| 5.4.2 | Eager Protocol | 117 |
| 5.4.2.1 | Eager Sender Side Alternatives: | 117 |
| 5.4.2.2 | Eager Receiver Side Alternatives: | 119 |
| 5.4.2.3 | Non-blocking Send/Recv Eager designs: | 120 |
| 5.4.3 | Rendezvous Protocol | 121 |
| 5.5 | Performance Evaluation | 123 |
| 5.5.1 | Experimental Testbed and System Software | 123 |
| 5.5.2 | Micro-benchmark Results | 124 |
| 5.6 | Related Work | 131 |
| 5.7 | Summary | 132 |</p>
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.2</td>
<td>Active Synchronization Scenarios</td>
<td>172</td>
<td></td>
</tr>
<tr>
<td>7.4.3</td>
<td>Passive Synchronization Scenarios</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>7.4.4</td>
<td>Hardware Limitation Scenarios</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>7.4.5</td>
<td>Non-contiguous Datatype Usage Scenarios</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>7.4.6</td>
<td>Combination of synchronization methods</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>7.5</td>
<td>Experimental Evaluation</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>7.5.1</td>
<td>Experiment Testbed</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>7.5.2</td>
<td>Application-kernel/Proxy-Application Evaluation</td>
<td>178</td>
<td></td>
</tr>
<tr>
<td>7.5.2.1</td>
<td>Related Work</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>7.5.2.2</td>
<td>Summary</td>
<td>181</td>
<td></td>
</tr>
<tr>
<td>7.6</td>
<td>Impact on the Design and Use of MPI Libraries on GPU Clusters</td>
<td>182</td>
<td></td>
</tr>
<tr>
<td>7.7</td>
<td>Impact on the Design and Use of MPI Libraries on Xeon Phi Clusters</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>Impact on the Design and Use of Energy Efficient MPI Applications</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>7.9</td>
<td>Software Release and Wide Acceptance</td>
<td>184</td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td>Future Research Directions</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td>8.1</td>
<td>Heterogeneity-Aware Data Movement for HPC Architectures, Deep Learning Systems, and Big Data Frameworks</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td>8.2</td>
<td>Exploiting Control-Plane Decoupling Techniques for Collectives, One-Sided MPI operations, and for Task-Parallelism on GPU Clusters</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>8.3</td>
<td>Energy-Aware MPI (EAM) to Address Energy Usage in Big Data Frameworks</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>Conclusion and Contributions</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>193</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Relative Point-to-Point latency between different locations, L - Local, R - Remote</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>Notation used to identify different collectives</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Semantics of Stream-based MPI Communication</td>
<td>111</td>
</tr>
<tr>
<td>5.2</td>
<td>Acronyms of the GDS-based point-to-point MPI protocols</td>
<td>116</td>
</tr>
<tr>
<td>6.1</td>
<td>Symbols used for communication and energy modeling of proposed MPI runtime</td>
<td>134</td>
</tr>
<tr>
<td>6.2</td>
<td>Summary of energy efficiency results with EAM runtime. Performance degradation and energy improvements are calculated using optimistic execution as baseline</td>
<td>146</td>
</tr>
<tr>
<td>7.1</td>
<td>Symbols used for communication and energy modeling of proposed MPI runtime</td>
<td>164</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Presence of Multiple Communication Paths in Heterogeneous Setups [55]</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Varying Communication Costs Along Different Paths in Heterogeneous Setups</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Research Framework</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Communication Stack in Xeon Phi [55]</td>
<td>30</td>
</tr>
<tr>
<td>2.2 Peak inter-node bandwidth for different paths on MIC clusters [55]</td>
<td>33</td>
</tr>
<tr>
<td>2.3 Peak inter-node bandwidth for different paths on GPU clusters [55]</td>
<td>34</td>
</tr>
<tr>
<td>3.1 Point-to-Point communication levels in a MIC cluster</td>
<td>37</td>
</tr>
<tr>
<td>3.2 Recursive doubling allgather algorithm</td>
<td>40</td>
</tr>
<tr>
<td>3.3 Ring-based allgather algorithm</td>
<td>43</td>
</tr>
<tr>
<td>3.4 Dual-Ring-based Allgather</td>
<td>44</td>
</tr>
<tr>
<td>3.5 Pairwise Exchange Alltoall algorithm</td>
<td>49</td>
</tr>
<tr>
<td>3.6 Performance of Small Message MPI_Allgather</td>
<td>53</td>
</tr>
<tr>
<td>3.7 Performance of Large Message MPI_Allgather</td>
<td>55</td>
</tr>
<tr>
<td>3.8 Performance of Small Message MPI_Alltoall on 16 Xeon Phi Nodes</td>
<td>56</td>
</tr>
<tr>
<td>3.9 Performance of Large Message MPI_Alltoall on 16 Nodes</td>
<td>57</td>
</tr>
</tbody>
</table>
4.17 Overlap of MPI_Ialltoall ............................................. 89
4.18 Latency of MPI_Ialltoall ............................................. 90
4.19 Overlap of MPI_Isscatter ............................................ 91
4.20 Latency of MPI_Isscatter ............................................ 92
4.21 Overlap of MPI_Igather ............................................. 93
4.22 Latency of MPI_Igather ............................................. 94
4.23 Overlap of MPI_Ialltoall, MPI_Isscatter, MPI_Igather with 64 nodes (2PPN) . 95
4.24 Latency of MPI_Ialltoall, MPI_Isscatter, MPI_Igather with 64 nodes (2PPN) . 96
4.25 Effect of Compute Location on Latency/Overlap ..................... 97
4.26 Latency Comparison with blocking variant .......................... 97
5.1 Pseudocode for 2DStencil CUDA+MPI Test ............................ 103
5.2 Offload Overheads dominate with strong scaling as CPU-GPU synchronization becomes more frequent ............................ 104
5.3 Blocking Stream-based Send/Recv Semantics .......................... 106
5.4 Extended CUDA/InfinBand software stack for GPUDirect aSync ...... 112
5.5 GPU architecture (Courtesy: NVIDIA) .................................. 114
5.6 Eager Design ......................................................... 117
5.7 Sender-Side Rendezvous Design ...................................... 122
5.8 Receiver-Side Rendezvous Design .................................... 123
5.9 Throughput of Registration-based Eager Send/Receive + Kernel Call followed by Stream-completion Call .................. 126
Chapter 1: Introduction

The field of high-performance computing (HPC) has been undergoing a revolution over the past decade with the culmination of Dennard’s scaling. The two most notable changes that have stemmed from this development have been 1) the proliferation of many-core architectures (e.g., PCIe devices such as NVIDIA GPUs [32] and many-core devices such as Intel MICs [17]), and low-power processors that can operate at a near-threshold voltage (NTV) [26]; 2) the realization of novel commodity network subsystems that provide low-latency (< 1µs) and high-bandwidth (100Gb/s on Enhanced Data Rate or EDR InfiniBand) but also have the additional capabilities such as a) the ability to directly access many-core PCIe device memories; b) the ability to orchestrate a sequence of enqueued communication operations, trivial computation operations, and dependency satisfaction primitives; and c) being able to service network requests directly from other PCIe devices (such as GPUs) with minimal CPU assistance. An increased usage of such devices can be seen in the November’16 TOP500 list, where over 100 machines are composed of heterogeneous processing entities while nearly 50% of the machines are built using asynchronous and Remote Direct Memory Access (RDMA) capabilities of InfiniBand (IB) or InfiniBand-like interconnects. Furthermore, three of the top five TOP500 machines in the November ’16 list are composed of either many-core processors such as Intel MICs or NVIDIA GPUs [5].
These changes have come about with a two-fold end goal of minimizing both execution time and energy expenditure of scientific applications that use these systems.

To reduce execution time of compute-phases of applications, HPC systems have increasingly relied on the high degrees of parallelism available on many-core architectures. While these many-core devices bear the potential for higher application throughput and larger throughput-per-watt, they come at the cost of rendering HPC systems heterogeneous. Many-core NVIDIA and Intel MIC architectures generally have separate memory and network subsystems and also have different processor capabilities in comparison with native host processors that house these devices. As a consequence, this introduces differential communication and computation costs at varying levels. For instance, the rate at which the network interface can read data from the GPU memory may be less than one-sixth the rate at which it can read from the host processor’s memory. Another difference is the rate at which cores operate on these many-core architectures (between a third and half the host processing rate), which affects the rate at which messages can be sent out from the node. Such intrinsic differences in communication and compute-capabilities can adversely degrade the performance of communication software stacks that frequently leverage these underlying hardware subsystems for different purposes. To alleviate some of the problems associated with heterogeneity, network subsystems that serve as the interconnection of several many-core nodes are being designed with a range of features. These include RDMA and multicast capabilities directly to/from device memory, network offload capabilities, and the ability to service requests originating from other PCIe devices such as GPUs and other accelerators. Hence, new communication designs that take advantage of a plethora of novel network features are needed for better communication throughput, overlap opportunities, and to allow for control plane decoupling from the CPU.
To reduce energy expenditure in the HPC systems, existing general-purpose processors are being equipped with various energy and power throttling capabilities. Mechanisms such as dynamic voltage frequency scaling (DVFS) [60] and core-idling/blocking [35] are commonly used in practice to control power and energy usage of applications. However, as the usage of these mechanisms incur high overheads, these power/energy knobs or gears must be used sparingly or only when there is a sufficient opportunity to apply them without notable performance impact. Frequent usage potentially leads to bad application performance and under-usage leads to low power/energy savings. Hence, the challenge generally lies in identifying the right instances to apply these gears.

Traditionally, communication substrates such as Message Passing Interface (MPI) [7] and Partitioned Global Address Space (PGAS) [75] have been used to realize parallel implementations of scientific applications in distributed memory environments. With increasing use of many-core architectures, these communication substrates are used in tandem with compute-offload models such as OpenMP [13] (eg: MPI+OpenMP) and CUDA [32] (eg: MPI+CUDA), where data needs to be moved to the site of many-core processors for the acceleration of compute phases. Subsequently, this has given rise to adaptation of MPI/PGAS models where data buffers that are located on the many-core device’s memory are directly specified in communication calls (eg: CUDA-aware MPI [74]). From an application programmer’s perspective, this avoids the need to stage data located on many-core devices on to host memory before communication calls are made. It is the ability to directly specify device buffers in communication calls, the ability to easily express common communication patterns (such as collectives), and benefit from associated performance portability that makes the use of MPI/PGAS routines favorable in distributed realization of scientific applications.
Owing to the wide usage of MPI and PGAS as a communication substrate and due to the fact that communication phases often dictate the overall performance of typical large-scale applications (e.g., HPCG [15]), it is critical to design MPI/PGAS runtimes that leverage the underlying hardware in a near-optimal-manner and satisfy traditional latency, throughput, and overlap requirements of applications, especially in heterogeneous environments.

Most state-of-the-art MPI communication protocols and algorithms that exist within literature assume uniformity in the cost of transferring data along different communication paths as well as in the associated network processing compute costs at different endpoints. In heterogeneous setups, however, there are a multitude of paths and different network subsystems with varying capacities (Figure 1.1 and Figure 1.2). Popular dense collective communication algorithms, such as Pairwise-Alltoall [67] and Ring-Allgather [68] for instance, that treat all communication paths equally, tend to make repeated use of slower communication paths and incur severe penalties as a result. Hence, it is critical to propose adaptations to these algorithms that differentiate costs along different paths for good performance. This raises the following question: “Can heterogeneity-aware communication algorithms be proposed such that they overcome uniformity assumption drawbacks of state-of-the-art MPI collective algorithms?”. In addition to adaptations, it is also important to leverage additional special capabilities of network subsystems that accompany these heterogeneous devices. For instance, on Intel MIC architectures it is possible to issue asynchronous transfers between the MIC’s memory and the native host memory over an internal channel called SCIF [2]. This allows for added overlap opportunities between multiple transfer options in collective scenarios. Hence, the following question arises: “Can new heuristics be proposed and can algorithms that leverage novel asynchronous capabilities be designed for the realization of efficient heterogeneity-aware collectives?”
1. Intra-MIC
2. Intra-IOH MIC-MIC
3. Inter-IOH MIC-MIC
4. Intra-socket MIC-Host
5. Intra-socket
6. Inter-socket
7. Inter-node
8. Inter-node MIC-MIC
9. Inter-node MIC-Host

Figure 1.1: Presence of Multiple Communication Paths in Heterogeneous Setups [55]

Figure 1.2: Varying Communication Costs Along Different Paths in Heterogeneous Setups
From a more purely network-centric perspective, the development of novel software infrastructures that allow for direct memory transfers between PCIe-peer devices offer the potential opportunity to avoid high latency paths induced by data staging paths and allow for the realization of truly asynchronous collective operations issued from device buffers. In addition, such capabilities bear the potential of being combined with existing multicasting [34] or network offloading [65] features present in popular InfiniBand implementations. This raises the question: “Can novel MPI collective designs that take advantage of network features such as multicasting in heterogeneous setups be proposed to achieve high throughput while overcoming reliability limitations of multicasting?” These are essential in throughput-critical streaming applications where broadcast operations are frequently used and is often a performance dictator. In addition, when non-blocking collective operations need to be issued directly from PCIe device memory (as in the case of CUDA-aware MPI programming), it is desirable to not involve the CPU for better overlap and resource utilization. Therefore, we ask: “Can novel designs be proposed that take advantage of direct PCIe peer-to-peer communication and the network controller’s ability to execute a sequence of low-level communication primitives to realize high performance offloaded non-blocking collective operations from PCIe device memory?”

While exploitation of PCIe peer-to-peer transfer methods can lead to high-performance communication operations from device buffers, they still need CPU intervention in synchronizing communication and compute phases. In other words, the CPU needs to ensure completion of either compute or communication phase before initiating the other if there exists a dependency between the two phases. Therefore, to further reduce the involvement of the CPU, novel network software infrastructures have been developed that allow GPU devices to directly issue network operations with minimal CPU assistance. This allows
for greater control plane decoupling and reduction of synchronization overheads that are necessary when synchronization cost begin to dominate execution time. In general, this raises the following two challenges: “Can novel software infrastructures be leveraged to allow PCIe devices such as GPUs to directly issue network operations upon completion of compute kernels and can new kernels resume upon network completion without CPU assistance? Can efficient CPU-GPU Control plane decoupling be realized while retaining easy-to-use MPI programming semantics?”

In addition to achieving good communication performance, as applications spend a fair amount of time in communication routines, energy expended by the CPU during such communication phases must be saved without affecting application performance. For such methods to be widely applicable, these savings must be made without needing application hints or code changes. Communication routines, predominantly, have been designed with high-performance expectations thus far without energy savings in mind. The energy spent by the CPU during communication phases, nevertheless, remains one of the biggest contributors of overall energy expenditure of an application. In real-world applications, load-imbalance and system effects (such as OS-jitter, network congestion, etc) present sufficient opportunities (slack) to conserve energy without degrading the performance of communication operations. Existing approaches use black-box approach of applying energy-levers at the cost of communication performance penalty; or use ad-hoc methods that suffer from lack of generality for better exploitation of energy saving opportunities [23, 66]; or make temporality assumptions about communication behavior that can lead to either sub-optimal savings or communication performance penalties [58]. Saving energy during long communication routines in an application-oblivious manner with minimal performance impact is critical but it is necessary to achieve this without the drawbacks of previous works. This
raises the question of “Whether intimate knowledge of communication protocols of communication routines can be exploited to propose generic online methods that conserve energy at appropriate moments in communication phases of both 2-sided and 1-sided communication routines without suffering performance degradation and without application hints?”

1.1 Problem Statement

In order that applications continue to exploit underlying hardware in near-optimal manner (regardless of whether the system setup is heterogeneous or not) and to ensure energy efficient execution without sacrificing communication performance under varying circumstances, the questions raised in the previous section (Chapter 1) must be addressed. For this, the communication runtime must be cognizant of differences in the cost of transferring data across different paths and must minimize on the use of expensive paths in communication calls if possible. State-of-the-art collective algorithms must be made to adapt accordingly. Furthermore, algorithms that exploit special capabilities of these new transfer channels must be devised to ensure better overlap among different communication steps involved in a collective operation. Novel network software infrastructure must be leveraged for a better realization of collective operations that yield better throughput and better asynchronous progress in streaming and non-blocking collective scenarios respectively. Mechanisms that allow PCIe devices such as GPUs to issue network operations must be leveraged to alleviate common synchronization overheads in strong scaling scenarios. Lastly, the communication runtime must identify within a communication call when to apply energy levers such that the instances of application that lead to energy savings without performance degradation (true positive) are maximized and instances of application that lead to performance degradation with or without energy savings (false positives) are minimized. Hence, an intimate
knowledge of 2-sided and 1-sided communication protocols need to be exploited in order to yield a generic and an energy-efficient communication runtime that saves energy with negligible performance degradation. This dissertation attempts to improve communication performance in heterogeneous setups as well achieve energy savings during communication routines without perturbing application performance by asking the following questions:

1. Can traditional MPI collective algorithms be modified by proposing heterogeneity-aware communication algorithms that overcome the drawbacks that accompany uniformity assumptions?

2. Can new algorithms be proposed that better exploit network subsystems on heterogeneous nodes for the design of high-performance dense MPI collectives?

3. Can novel designs be proposed that take advantage of special-capability network subsystems that have peer-to-peer access with other PCIe devices as well the ability to execute a list of network tasks to realize high throughput broadcast operations and offloaded non-blocking collective operations from PCIe device memory respectively?

4. Can designs that leverage the abilities offered by novel network infrastructures to directly issue communication operations from the GPU be realized for better CPU-GPU control plane decoupling? Can synchronization overheads be reduced? Can MPI-like semantics be retained while honoring the GPU’s CUDA stream semantics and can efficient protocols be devised?

5. What kind of performance benefits can be seen using heterogeneous architecture-aware communication routines with scientific kernels and applications?
6. Can methods that leverage communication protocol knowledge be proposed to design an application-oblivious energy-aware MPI runtime that saves energy during 2-sided point-to-point and collective routines?

7. Can opportunities for saving energy in communication phases that access memory remotely through 1-sided operations be identified? Can rules for automatically saving energy in both 2-sided and 1-sided communication contexts be proposed?

8. What are the energy savings that can be seen across MPI applications/proxy-applications from the use of an application-oblivious energy-efficient communication runtime? How does its usage affect application performance?

1.2 Research Framework

Figure 1.3 depicts the research framework that this dissertation uses to address the problems stated in Chapter 1.1. Here, we discuss how we use the framework to address each of the challenges in detail.

Can traditional MPI collective algorithms be modified by proposing heterogeneity-aware communication algorithms that overcome the drawbacks that accompany uniformity assumptions?

MPI collectives are useful for expressing group communication in distributed scientific applications and are commonly used for many important application kernels. Often, they dictate the overall performance of MPI applications. Traditional MPI collective algorithms such as binomial/knomial-tree broadcast/reduce, hypercube algorithm for allreduce, recursive-doubling allgather, Bruck’s alltoall, etc are popular in practice owing to their asymptotic efficiency and due to the ease of implementation. These algorithms, however, have assumed uniformity in the cost of traversing different links connecting MPI ranks.
Figure 1.3: Research Framework
(which generally translate to links connecting different nodes), in accessing different memory banks of potentially different capabilities, and in the assumption of uniformity in processing capabilities of different compute units involved. These assumptions do not hold true on existing and upcoming clusters with many-core architectures, where links connecting data associated with different MPI ranks are of different capacities and also both the processing and memory capabilities of different MPI ranks vary based on the location of the core on which the MPI rank resides or in the choice of memory used for that particular communication routine. In this research framework, we propose ways of modifying popular collective algorithms (especially dense collective algorithms such as MPI_Allgather and MPI_Alltoall) by identifying the bottlenecks associated with cost uniformity assumptions and proposing architecture-independent generic methods to alleviate the drawbacks of these assumptions. These modifications are such that they closely retain the asymptotic properties of the original algorithm but allow for efficient overall costs by avoiding expensive paths or by minimizing use of said expensive paths.

Can new algorithms be proposed that better exploit network subsystems on heterogeneous nodes for the design of high-performance dense MPI collectives?

While adaptations of traditional MPI collective algorithms allow for higher performance in comparison with their original counterparts, they do not fully exploit novel features that accompany the network subsystems that are present on modern heterogeneous clusters. Furthermore, these adaptations are not cognizant of the topology that interconnects specific subsystems that constitute to the entire MPI job. For instance, on Intel MIC KNC architecture, the coprocessor and the native host, that houses the PCIe device, are interconnected by the SCIF channel which is capable of executing asynchronous DMA transfers between host and device memory 2.2.3. In addition, within the KNC coprocessor,
there exists a ring topology that interconnects different cores of Intel Xeon Phi. We propose to leverage both topological knowledge and novel transfer mechanisms in order to design efficient algorithms and new heuristics for dense collective operations such as allgather and alltoall.

Can novel designs be proposed that take advantage of special-capability network subsystems that have peer-to-peer access with other PCIe devices as well the ability to execute a list of network tasks to realize high throughput broadcast operations and offloaded non-blocking collective operations from PCIe device memory respectively?

CUDA-aware MPI programming is becoming the de facto method for writing distributed scientific applications that leverage the GPU’s computing power through the CUDA programming model. Traditionally, CUDA-aware MPI programming meant that the runtime stages GPU data on to host memory behind the scenes without the user explicitly needing to stage data transfers (either on a per-use basis or in a pipelined manner). This is carried out by the runtime prior to issuing network operations in order to realize end-to-end GPU data transfer over MPI calls. Modern GPU clusters, however, are being equipped with a plethora of network software novelties that allow for direct peer-to-peer transfers between any two PCIe devices that constitute a single node. GPU Direct RDMA (GDR) is a realization of this ability that allows the NIC/HCA to directly read/write to/from GPU memories and thus remove the necessity of staging data on host memory prior to transferring it over the network. While this technique has been used in the context of point-to-point transfers, its potential in the collective context is unexplored. In this dissertation, we specifically look at combining HCA/NIC’s ability to directly access GPU memory with existing features such as hardware multicast and offloading technology (such as CORE-Direct) in order
to design efficient broadcast operations in streaming scenarios and in order to propose and
design GPU-buffer based non-blocking collectives respectively.

Can designs that leverage the abilities offered by novel network infrastructures to
directly issue communication operations from the GPU be realized for better CPU-
GPU control plane decoupling? Can synchronization overheads be reduced? Can
MPI-like semantics be retained while honoring the GPU’s CUDA stream semantics
and can efficient protocols be devised?

Recently, mechanisms have been developed that enable issuing network operations di-
rectly from GPU devices, with minimal CPU assistance. This bears the potential of de-
coupling the CPU-GPU control plane and hence reduce inherent CUDA synchronization
overheads that the CPU generally has to oversee as it tries to satisfy dependencies between
compute and communication phases. To leverage these mechanisms, a two-fold approach
that involves 1. Proposing communication operations with semantics that either closely
resembles popular MPI semantics or those that minutely change MPI semantics are needed
while still honoring CUDA stream ordering semantics 2. Designing protocols that honor
both MPI and CUDA stream semantics while providing good performance are critical. In
this dissertation, we take advantage of a novel software framework called the GPUDirect-
Async to prepare work requests efficiently on the CPU and issue network operations from
GPU buffers along with GPU kernels in a batch-fashion according to CUDA stream order.
This is done in a way such that network operations are issued when all preceding opera-
tions complete or when semantics allow these operations to be issued. Likewise, designs
to ensure that CUDA kernels execute only upon communication dependency satisfaction
are also proposed. In addition, an MPI application-mimicking kernel is redesigned with
minimal changes in order to demonstrate the gains offered by the proposed extended-MPI operations.

Can methods that leverage communication protocol knowledge be proposed to design an application-oblivious energy-aware MPI runtime that saves energy during 2-sided point-to-point and collective routines?

Energy and power are first-order priorities on modern HPC clusters. Since the inception of power/energy throttling mechanisms on processors (such as DVFS and core-idling), researchers have devised different methods to conserve energy in the context of distributed scientific applications. As applications spend a significant amount of time in communication (especially at large scales), communication routines have been targeted for energy savings owing to the lack of actual compute work during these communication phases. The simplest of these methods use a black-box approach, where a switch to low-energy gear is made at the start of each MPI call and a return to a higher energy gear is made upon returning from the call. This approach is portable but can adversely affect performance, especially when the application is dominated by short message transfers and because often there is fairly high overhead in switching energy gears. Other methods such as Adagio [58], make a more intelligent decision about switching energy gears. Specifically, Adagio predicts the time associated with making an MPI call and based on a previous run decides whether or not to switch energy gears when making the MPI call. This allows for critical path identification and works best when the critical path is fairly stable. A pathological case is when the critical path change at each instantiation of the call which can lead to many false positives and hence performance degradation. Essentially, methods used by Adagio works best in iterative applications with temporal reproducibility in the behavior of MPI calls. On most HPC systems, however, systems effects like OS-jitter, network-congestion
and lack of load-balance break reproducibility assumptions even in seemingly iterative and
 temporal applications. In this framework, we propose ways of conserving energy during
 MPI calls that take a closer look at the protocols used to realize MPI calls and together
 with logGP model, makes an online decision of whether or not switch energy gears in an
 application-oblivious manner and with minimal performance degradation.

Can opportunities for saving energy in communication phases that access mem-
ory remotely through 1-sided operations be identified? Can rules be generated for
automatically saving energy in both 2-sided and 1-sided communication contexts be
proposed?

One sided semantics allows applications to decouple data movement from data syn-
chronization. One-sided transfer operations (such as Put, Get, and Accumulate) together
with synchronization operations (such as Fence, Lock, and Unlock) are increasingly being
used by applications. MPI-3.0 standard provides for a complete set of Remote Memory
Access (RMA) calls that have been adopted by applications and high-level communication
runtimes such ARMCI, OSHMPI alike owing to the bare-metal performance that imple-
mentations provide. Irregular applications benefit from the use of RMA calls and spend a
significant portion of their execution time in RMA phases. Hence, saving energy during
RMA phases can lead to overall energy savings in applications that directly take advan-
tage of MPI-3 1-sided calls as well as in applications that indirectly make use of it through
high-level communication runtimes. While attempts to study the performance of 1-sided
operations have been made, their impact on the energy of the entire application has not
been well studied. To add to this, the flexibility in semantics that the MPI standard offers
in implementing RMA operations make reasoning about energy expenditure of these calls
complex. This dissertation studies interactions that RMA calls can have with the application and proposes rules for automatically saving energy during RMA phases with minimal performance degradation.
Chapter 2: Background

2.1 InfiniBand

InfiniBand (IB) is a switched fabric standard designed for interconnecting compute and storage (I/O) nodes in High-End Computing (HEC) systems [1]. Owing to its low-latency, high bandwidth, and CPU low-overhead it is commonly used as an interconnect for commodity clusters including many in the TOP500 list of supercomputers. The list released in November ’16 shows that 187/500 systems use IB.

2.1.1 IB Verbs and Available Transports

Communication between endpoints interconnected by IB is achieved using a set primitives called Verbs. This is equivalent to sockets in Ethernet parlance. Connection establishment is generally achieved using Ethernet sockets. IB uses a twin queue-pair (one for sending and another for receiving) based model. Processes can enlist a set of network operations (work requests) and issue to a queue-pair (QP) either as `ibv_post_send` or `ibv_post_recv` to send and receive messages which is then executed by hardware. Unlike with sockets, a registration mechanism allows the user process to register a part of its memory with the IB network interface controller (NIC) which then allows the NIC to directly access as part of network transmissions. This is due to the fact that registration yields a pair of keys (local
and remote key) which can be used by local and remote processes to directly access said registered memory and thus realize operations such as direct remote read (or RDMA read) and direct remote write (or RDMA write) operations. During the time of creating the QP, an entity called the completion queue (CQ) must be associated with the QP for signaling completion of receive/send work requests. Completion of these requests results in the creation of completion queue entries (CQE) which can be queried (using `ibv_poll_cq`) for specific details concerning the completed request.

**IB provides reliable and unreliable data transfer through Reliable Connection (RC) and Unreliable Datagram (UD) transport modes respectively.** Both RC and UD transport is used in the experiments presented in this dissertation. RC provides end-to-end reliable data transfer and also ensures ordering of messages. This comes at the cost of requiring a QP for every other endpoint from one endpoint’s perspective. UD, on the other hand, guarantees neither reliability nor ordering but supports hardware multicast operation once multicast groups have been setup [34]. Apart from this, other difference between RC and UD is that UD’s maximum transmission unit (MTU) is between 2,048 and 4,096 depending on the IB vendor which has packetization implications for larger messages.

### 2.1.2 Remote Direct Memory Access (RDMA) using InfiniBand

In addition to supporting socket operations such as `send` and `receive` (under channel semantics), IB also supports memory semantics under RC transport. Unlike channel semantics, where both the sender and receiver are involved in the transfer of data between the pair, memory semantics allow data to be accessed (read/write) by a remote process with no intervention from the local process after an initial exchange of registration keys associated with local user memory. A remote process may issue an RDMA-write operation to directly
write to local memory or an RDMA-read operation to read directly from local memory without the software intervention of the local process (the NIC is still likely to participate). This allows zero-copy transfers to occur between two endpoints should the remote addresses are predetermined. Furthermore, no receive work requests are consumed from the perspective of the local process if RDMA operations are issued by a remote process except under special circumstances.

2.1.3 Hardware Multicast Capability using InfiniBand

IB Multicast feature can be used to realize data broadcast from one endpoint to multiple endpoints with one single operation. IB supports multicast operation on the UD transport alone and hence reliability and ordering must be undertaken by upper-level software layers. The multicast operation is achieved through multiple steps using IB. First, processes subscribe to a multicast group. For this, a multicast group must be setup using global IDs (GID) of NICs using InfiniBand management datagrams (MAD). From this, a Multicast GID (MGID) is obtained from the subnet manager. Second, the source of multicast must post a send operation and others must issue a receive operation using `ibv_post_send` and `ibv_post_recv` respectively.

2.1.4 Scatter-Gather-Lists (SGLs) in InfiniBand

InfiniBand provides `Scatter-Gather-Lists` convenience abstractions to perform data transfers from multiple non-contiguous user memory locations using one work request. In essence, this avoids the necessity of posting multiple send operations at the sender side and corresponding multiple receive operations at the receiver end. The underlying implementation is expected to combine (`gather`) data from different memory locations and disperse (`scatter`) them into different locations at the receiver endpoint. For channel semantics the
scatter and gather specification occurs at both receiver and sender respectively. In memory semantics, either scatter or gather is specified at the issuing end depending on whether the operation is an RDMA-read or RDMA-write respectively.

2.1.5 Offloading Network Operations in InfiniBand

Along with all of the standard InfiniBand features, IB vendors like Mellanox offer the ability to offload network activity entirely to the NIC/HCA beginning from ConnectX-2 series using a feature called CORE-Direct [65]. Using CORE-Direct, a list of send, receive, wait and primitive compute operations can be packed into one work request over RC transport mode using channel semantics and posted to the HCA. Later, the NIC independently progresses the list of enqueued operations and eliminates the constant need for the host processor to progress the communication tasks individually thus freeing up the CPU and allowing for potentially much better computation and communication overlap. Using such lists, non-blocking collective operations may be designed by upper-level libraries.

2.2 Programming Models for High Performance Computing Scientific Applications

This dissertation focuses on the MPI programming paradigm used as a communication substrate not just by applications directly but solver libraries (eg: P3DFFT [49]), high-level communication libraries (eg: ARMCI-MPI [42]) and other PGAS library realizations (eg: OSHMPI [20], MV2-X [3]).
2.2.1 MPI

Since its inception nearly twenty years ago, MPI has gained wide popularity as the most commonly used communication substrate owing to the availability of high-performance implementations over most available communication networks as well as performance portability factor that accompanies it. The latest update in the MPI standard, MPI-3.1, supports two-sided and one-sided point-to-point operations, and blocking and non-blocking collective operations as part of data transfer functions along with a list of convenience and setup functionality. Communication between endpoints (referred as a rank in MPI) is realized over high-performance networks using lower-level functionality offered by standards such as InfiniBand (As of MPI-3.1 standard a rank can have at most one endpoint). Thus performance achieved on one InfiniBand cluster is potentially ported to another IB cluster with little to no effort. Send/Recv operations or one-sided operations such as Get/Put can be used to transfer data between two ranks while group communication can be realized using common collective operations such as broadcast, all-broadcast, allreduce among others. Furthermore, non-blocking variants of Send/Recv and collective operations can also be issued and progress can be tracked using Wait/Waitall variations and using Fence/Flush-like operations for one-sided operations.

The MVAPICH2 MPI Library MVAPICH2 [41, 48], is an MPI implementation of the MPI-3.1 specification over InfiniBand. MVAPICH2 achieves near-peak latency and bandwidth that the underlying InfiniBand hardware adapter while providing good scalability and fault tolerance. The software is used by more than 2,700 organizations worldwide in 83 countries and is powering some of the top supercomputing centers in the current TOP500 list, including the 1st ranked Sunway TaihuLight system.
MVAPICH2 uses an RDMA-based *eager* protocol called RDMA-Fast-Path [36] or uses send-recv based operations 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 avoid copy costs that accompany eager protocol and to achieve higher 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 hierarchical and shared-memory based designs. It also provides tuned collective algorithms based on the message, job size and based on network and processor architecture detection. MVAPICH2 implements one of most optimized implementations of MPI one-sided communication by directly taking advantage of IB RDMA to implement Put, Get and certain atomic operations in both active and passive synchronization scenarios [30].

**Blocking MPI Collective Algorithms and Optimizations** Collective operations are used to express group communication patterns and are very commonly used across parallel applications. Asymptotically efficient collective algorithms using LogP, PlogP, and LogGP models have been proposed in literature and are commonly used to realize collective operations in popular MPI implementations such as MPICH2, OpenMPI and MVAPICH2 [52]. Rooted operations such as broadcast and reduce are most commonly realized using tree-based implementations in $O(\log N)$ time. Tree-based counterparts for gather and scatter however still require $O(N)$ time due to either doubling or halving of message size as messages traverse across edges of the tree. Allreduce operations, where the reduced data is available at all sites can be realized in $O(\log N)$ time using the hypercube algorithm. The Allgather collective operation is also known as the Alltoall-Broadcast and it involves each
process broadcasting data to every other process in the group. If there are N processes involved, the MPI_Allgather operation can be visualized as N concurrent broadcast operations with each process taking turns to act as the root. Small message MPI_Allgather operations are realized through the recursive-doubling algorithm in $O(\log N)$ time while the ring-exchange algorithm is used for large message allgathers in $O(N)$ time. All-to-all personalized message exchange is a dense personalized collective where each process exchanges a personal message unit with every other process. Bruck’s algorithm and pairwise exchange are used to implement short and large message All-to-all in $O(\log N)$ steps and $O(N)$ steps respectively.

MVAPICH2 uses a two-level communicator system to implement various collective operations in a hierarchical manner [24]. Processes that share the same address space (within a node) are encapsulated within the shmem communicator. The process with the lowest rank in each node is designated as the leader and a leader communicator includes all such leader processes, across all the nodes. Further, the library relies on advanced schemes to optimize the inter-node and the intra-node phases of the collective operation, often overlapping the two phases.

**Non-Blocking MPI Collectives** Non-blocking collectives were introduced with MPI-3 and are a natural extension of MPI-2 non-blocking point-to-point operations. They were primarily motivated by efforts to hide the execution time of long collective operations and also to absorb the impact of application load imbalance as well as system noise [51]. The challenge in using non-blocking collective lies in being able to identify program sections which are overlappable from an application perspective. From a runtime perspective, the challenge is in progressing as much of the collective call’s components without requiring CPU assistance to improve overlap factor. The algorithms used are similar to blocking...
counterparts and often the simplest of algorithms are chosen to avoid complex shuffles that might require loopback operations or intermediate copies that may require the CPU’s intervention.

2.2.2 GPU Architecture and CUDA Programming Model

Device Architecture and Programming Model A GPGPU can be viewed as a data-parallel many-core system that has its own memory and network subsystem that is connected to the host processor as a PCIe device. In this work, we focus on the use of NVIDIA GPU architecture with the Compute Unified Device Architecture (CUDA) programming model [44]. Some of the latest Tesla architectural revisions of NVIDIA GPUs for HPC is comprised of over 7.1 billion transistors, delivering over 1 TFLOP of double precision throughput and up to 3x the performance per watt of a Fermi architecture. It may include up to 15 Streaming Multiprocessor (SMX) units and 6 64-bit memory controllers. Each of the SMX units can include 192 singleprecision cores, 64 doubleprecision units, 32 special function units(SFU), and 32 load/store units. Each SMX can have 64KB of configurable shared memory/L1 cache and a 48KB of read-only data cache. Tesla K20C features 1536KB of dedicated L2 cache and 5GB of DRAM. The SMX allows four warps (groups of 32 threads) to be issued and executed concurrently. Kepler’s quad warp scheduler can dispatch two independent instructions per warp in each cycle. Unlike Fermi, Kepler also allows double precision instructions to be paired with other instructions. Kepler also boasts of features like Dynamic Parallelism and HyperQ that are aimed at increasing the utilization of GPUs. Compute code that runs on the GPU is often called a CUDA kernel. Additionally, however, CUDA exposes several other functionalities to transfer data to/from the GPU, to
track the progress of specific operation on independent streams of execution and to check on device properties among other things.

**GPU Work Enqueuing and Ordering with CUDA Streams and Events**

The GPU internally has multiple command queues and a hardware scheduling unit (GPU scheduling unit or GSU) through which work is submitted and scheduled across the SMs. The work submitted to a command queue is scheduled only after completion of all previous work in that command queue. However, work in different command queues can be scheduled independently. These command queues are exposed in CUDA through software constructs called *CUDA Streams*. CUDA kernels or CUDA memory copies submitted using the same CUDA Stream are guaranteed to execute in order. Kernels and copies submitted to different streams can execute out of order but might be ordered if two CUDA streams are aliased onto the same command queue underneath. CUDA provides a way to create dependencies between two streams and hence between two sequences of work using software constructs called CUDA Events. An event record (cudaEventRecord) can be thought of as a semaphore release while an event wait (cudaStreamWaitEvent) as a semaphore acquire. cudaEventRecord issued on a stream records the event (releases the semaphore) in order with other work issued on that stream. Similarly cudaStreamWaitEvent blocks any following work submitted on the stream for an event to be recorded (blocks on a semaphore acquire). These two calls can be used to create dependencies between work issues on different CUDA streams.

**System Architecture** GPUs are connected as peripheral devices on the I/O bus (PCI express). Communication between GPU and host, and between GPUs within a node has been a performance bottleneck as it involves costly transfers over the PCIe bus. CUDA 4.0 introduced Unified Virtual Addressing (UVA) feature. This feature enables a process to
have a unified address space across main memory and device memories of GPUs connected within a single node. Data can be moved directly from one GPU to another using Peer-to-Peer (P2P) communication, by-passing main memory. However, communication between GPU buffers used by different processes has to go through main memory. NVIDIA has tried to address this shortcoming in CUDA 4.1 by supporting Peer-to-Peer communication between processes through \texttt{culpe*/cudaIpc*} interface. CUDA Inter-Process Communication (IPC) made it possible for different processes using different GPU devices on a single node to communicate between the device memories without involving main memory.

**GPUDirect RDMA (GDR) for GPU Clusters** As current generation GPUs from NVIDIA are connected as peripheral devices (PCIe devices) on the I/O bus (PCI express), communication between a GPU and its native host, and between two GPUs can occur over the PCIe bus. Starting from CUDA 5.0 [44], GPUDirect has been extended to allow third party PCIe devices (such as the NIC/HCA) to directly read/write data from/to GPU device memory. This feature, which is currently supported with Mellanox InfiniBand network adapters, is called GPUDirect RDMA (GDR). This provides a fast path for moving data in the GPU device memory on to the network that completely bypasses the host and avoids host memory staging requirement. Although 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 P2P exchanges on modern node architectures. This is an artifact specific to node architecture and can indeed occur between any two PCIe devices which can have P2P transmission between them. This issue severely limits the performance achieved by GPUDirect RDMA for large message transfers. Performance of both P2P write and read operations are even worse when the devices are connected to different sockets.
GPUDirect-Async (GDS) for GPU Clusters
NVIDIA and Mellanox introduced mechanisms to initiate network operations directly from within the GPU. The general idea is to use the CPU to prepare and queue communication work to the IB HCA but to delay triggering of the operation by not ringing the doorbell. The act of ringing the doorbell is converted into a list of primitives offered by the GPU Scheduling Unit (GSU) and queued onto a stream. This allows GPU to trigger the start of a communication operation in stream order when prior compute work on the stream completes. Similarly, the operation of polling for an IB completion is converted into GSU primitives and queued onto a stream. This blocks any later activity submitted on the stream until polling completion is notified. These operations are realized using a mid-level library, libgdsync, through calls such as `gds_stream_queue_send` and `gds_stream_wait_cq` which issue send operation in stream order and stalls the stream execution sequence until an event is generated in the completion queue that is passed as an argument. In addition some CUDA extensions that prevent the operations on the stream to progress or allows a memory region to be written with a specific value are realized through `cudaStreamWaitVal` and `cudaStreamWriteVal` respectively.

MVAPICH2 for GPU Clusters
Message Passing Interface (MPI) is the de facto standard for parallel application development in the HPC domain. MVAPICH2 is a popular open-source implementations of MPI on InfiniBand, 10Gigabit Ethernet/iWARP and RDMA over Converged Enhanced Ethernet (RoCE). MVAPICH2 unifies data movement from/to GPU and host memories through the standard MPI semantics (also known as CUDA-Aware MPI). This design was first proposed by Wang et. al in [74]. MVAPICH2 achieves this through the Unified Virtual Addressing (UVA) feature that is provided starting from CUDA 4.0. MVAPICH2 further optimizes the performance of inter-node GPU-to-GPU communication by pipelining transfers from GPU to host memory, host memory to remote host
memory over InfiniBand and finally from the remote host to destination GPU memory. The three stage pipeline results in a significant boost in communication performance between GPUs on different nodes. MVAPICH2 also provides multiple solutions for intra-node GPU-to-GPU communication between processes, using shared memory and CUDA Inter Process Communication [56]. Lastly, MVAPICH2-GDR leverages GDR technology to allow MPI operations issued from GPU buffers to be directly accessed by the NIC/HCA and provides short latency and good bandwidth in the short message range while overcoming the requirement to stage data on the host memory [54].

2.2.3 Intel MIC Architecture and Programming Model

The Xeon Phi is a coprocessor, also available as a PCIe device which comes equipped with 61 processor cores which 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. To support a large number of cores present on the Xeon Phi, there are 8 memory controllers which can deliver a theoretical bandwidth of up to 352 GB/s. To take full advantage of such computing power MIC offers the following compute modes [6]: 1) Offload mode; 2)Co-processor-only mode; 3) Symmetric mode. In offload mode, all the MPI processes lie either in the host or in the MIC with the other processor used as an accelerator. In the co-processor-only mode, Xeon Phi is used as a stand-alone compute node with all MPI processes running on the co-processor. In the symmetric mode, MPI processes can run on either the host or Xeon Phi (usually both). The program has to be cross-compiled for Xeon Phi before it can be executed on it. Due to this limitation, in symmetric mode, the MPI job needs to be launched using a Multiple Program Multiple Data (MPMD) model even though it is the same program running both on the host and
the MIC. This arises from the fact that all binaries which run on the Xeon Phi must be cross-compiled for compatibility with co-processor. Hence, the host machines and their corresponding binaries to be launched must be configured with MPMD mode of launching.

![Communication Stack in Xeon Phi](image)

**Figure 2.1: Communication Stack in Xeon Phi [55]**

**Communication Channels** To support the full spectrum of usage models for the MIC, there are three modes of inter-process communication which can use a combination of the shared-memory, SCIF, and IB-verbs based communication channels. For two processes residing on the MIC, POSIX shared memory is supported and complemented with multi-threaded `memcpy`. Alternatively, for processes residing either in the MIC or on the MIC and the host, Intel’s SCIF communication API may be used. SCIF is a sockets-like API for communication between processes on MIC and host within the same system. Much like the sockets approach, entities called end-point descriptors are bound to ports and connections are established using `scif_listen`, `scif_connect` and `scif_accept` routines. All ensuing data transport routines make use of these end-point descriptors to identify data sources and sinks. 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 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. This read and write can make asynchronous progress with calls such as `scif_readfrom` and `scif_writeto` respectively. To indicate the completion of RMA operations, SCIF offers many routines. `scif_fence_signal` waits for all outstanding operations to finish and indicates completion by writing a specified value into a specified destination in the registered space [6]. Completion is detected by polling on that address for its value to change.

Intel’s Manycore Platform Software (MPSS) for Xeon Phi provides two ways of using IB verbs for communication on MIC clusters, as depicted in Figure 2.1. 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 symmetric mode of communication on just the MIC or between the MIC and the host processor. This harnesses the advantages of the peer 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 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 IB even in the absence of a physical HCA as
all verbs operations are implemented using the SCIF interface and the communication goes directly over the PCIe. This is especially beneficial for MPI processes which reside on the Xeon Phi and provides ease of porting existing MPI applications to the MIC.

2.3 Architectural Bottlenecks with PCIe Devices

Communication operations between MPI processes whose communication buffers reside on PCIe devices at different compute nodes will involve the InfiniBand Host Channel Adapters (HCAs) transferring data across the network. The IB HCA is also involved in moving data when one process is on a MIC device and the peer process is on a remote host processor. Ideally, whether the HCA is reading data from the MIC or writing data to the MIC, the communication performance should be similar. However, we observe that the communication costs of these operations largely differ on state-of-the-art computing platforms based on Intel Sandy Bridge/Ivy Bridge/Haswell and Xeon Phi coprocessors. These differences are shown in Figure 2.2(a). The peak bandwidth achieved in a network transfer operation that involves the HCA reading from the MIC memory is just about 962 MB/s, when the HCA and the MIC device share the same I/O Hub. On the contrary, the corresponding peak bandwidth when the HCA is writing into the MIC memory is about 5,280 MB/s. Further, if the MIC device and the InfiniBand HCA are connected to different I/O Hubs, the corresponding read/write peak bandwidths are significantly lower. In contrast, Figure 2.2(b), shows the communication performance between MPI processes executing on the host processor and the MIC coprocessor, within the same compute node and between two host MPI processes located on two different nodes both of which are significantly better than with the HCA being involved in the transfer which indicates the peer-to-peer
bottleneck that exists on such systems. Owing to this behavior, on such heterogeneous systems, communication operations between MPI processes will not be able to achieve peak network bandwidth if the underlying MPI library is based on uniformity assumptions.

A similar problem is seen in GPU clusters. As GDR allows data from the GPU to be directly sent over the network without being staged through host memory, the HCA is again capable of directly reading and writing from/to GPU memory. Figure 2.3(b) compares the internode GPU-to-GPU peak latency and bandwidth of IB verbs level communication from GPU Device memory achieved using GDR with those when data is staged through the host for a Sandy Bridge machine (PCIe 3.0) equipped with a Mellanox IB FDR adapter and an NVIDIA Kepler K20 GPU. GDR offers very low latency compared to transfers staged through the host. However, its bandwidth is severely limited when compared to the bandwidth possible by staging through the host. Again, such limitations have to be taken into consideration while designing MPI libraries to effectively take advantage of GDR.

![Diagram showing peak inter-node bandwidth for different paths on MIC clusters](image)

(a) To/From MIC using the HCA  
(b) To/From Host and To/From MIC without HCA

Figure 2.2: Peak inter-node bandwidth for different paths on MIC clusters [55]
2.4 Levers for Energy Conservation in Applications and Power Monitoring

2.4.1 Power Levers

In this dissertation, the baseline power mode for running MPI library routines is polling mode where the CPU continuously polls on memory or on IB completion queue(s) (CQ), where send and receive completions are placed. This keeps the CPU in busy-wait state where there is close to full power expenditure because the underlying architecture assumes that an update in memory or on CQ justifies the need to not go into automatic power saving mode.

Another power mode available on InfiniBand and other high-speed networks is interrupt-driven execution. This lever achieves core-idling by relinquishing the CPU and blocking on a network event. When a send/receive completion occurs, the process is re-scheduled for computation. This power mode is effective if the overhead can be amortized over slack. For instance, for the TACC Stampede testbed used in this work, we observed that there is
an overhead of $5\mu s$ in calling using this mode. However, it was also observed that interrupt-driven execution reduces the power consumption by a maximum of 66% in comparison to the polling mode.

Another power mode which can be used is DVFS. In practice, this power lever requires super-user access, which makes it difficult (if not impossible) to be deployed/used on production systems, such as the one used in this work. While several homegrown clusters have reported success on a small-medium scale with DVFS, the super-user restriction makes it difficult to use DVFS on every cluster, unlike interrupt-driven execution which is available on all InfiniBand clusters. We considered several large scale production systems such as TACC Stampede, and PNNL Cascade. However, none of them provided DVFS support for general users. Due to their unavailability on production systems, the works in this dissertation has used interrupt-driven execution as the primary power lever.

2.4.2 Power Measurement

The aggregate power consumption of MPI libraries are measured using Intel Running Average Power Library (RAPL) interface [70]. RAPL provides fine-grained measurement of the CPU socket and memory power consumption. Work from this dissertation, built a thin measurement layer to collect power information on each socket (de-duplication), handle wraparounds by periodic accumulation ($\approx 30s$), and aggregate them at $MPI\_Finalize$ using reduction.
Chapter 3: Heterogeneity-Aware Dense Collective Designs for InfiniBand MIC Clusters

In this chapter, we present adaptations required to make traditional MPI algorithms efficient in the heterogeneous settings of InfiniBand MIC clusters. First, we develop a cost model to account for the differences in transferring data across various communication links in heterogeneous MIC clusters. Then, we describe heuristics used to leverage the knowledge of topological differences in network subsystem that interconnect Intel Xeon Phi cores as well the novel transfer mechanisms that allow better exploitation of network resources on such systems. Through these designs, we closely retain the asymptotic complexity of the original algorithms while improving on the overall execution time of collectives. Experimental evaluation shows that the adaptations proposed and the heuristics designed provide significant improvements in communication performance at scale when the Xeon Phi is operated in a symmetric-mode of execution (where MPI processes exist on both native host cores as well as on the Intel MIC architecture).

3.1 Cost Model

In traditional homogeneous clusters, there are only two levels of communications: intra-node and inter-node. Within clusters with co-processors, the levels of communications increase to eight: local host to local host, local host to remote host, local host to local
MIC, local host to remote MIC, local MIC to local host, local MIC to remote host, local MIC to local MIC and local MIC to remote MIC. In the following discussion, we use the following format $T_{\{source\}\{destination\}\{distance\}}$ to present the latency per byte between source and destination with distance, as shown in Figure 3.1. For example, $T_{MHR}$ means the latency per byte to send a message from a MIC to a remote host.

![Figure 3.1: Point-to-Point communication levels in a MIC cluster](image)

The MPI communication latencies for the eight different communication levels are shown the results in Table 3.1. The latencies of other seven communication levels are normalized based on the local host to local host latency. For small and medium message range, any communication involving MIC is 20 to 30 times of local host-to-host latency. When the message size increases, the latency for communication with local MIC becomes comparable with communication between local hosts. However, for remote MIC, the latency is still 17 times of the local host-to-host latency. These differences in communication performance serve as guidelines for designing heterogeneity-aware MPI_Allgather and MPI_Alltoall calls.
Table 3.1: Relative Point-to-Point latency between different locations, L - Local, R - Remote

<table>
<thead>
<tr>
<th></th>
<th>H-H</th>
<th>H-M</th>
<th>M-H</th>
<th>M-M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L</td>
<td>R</td>
<td>L</td>
<td>R</td>
</tr>
<tr>
<td>small(8)</td>
<td>1</td>
<td>6</td>
<td>21</td>
<td>23</td>
</tr>
<tr>
<td>medium(1K)</td>
<td>1</td>
<td>9</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>large(1M)</td>
<td>1</td>
<td>1.4</td>
<td>1.3</td>
<td>17</td>
</tr>
</tbody>
</table>

H - Host, M - MIC, L - Local, R - Remote

3.2 Heterogeneity-Aware Allgather Designs

The drawbacks that traditional MPI_Allgather collective algorithms encounter on MIC-like architectures are owing to their uniformity assumptions. These can be overcome with a knowledge of costs of different communication paths. Accordingly, we propose adaptations as well as new heuristics to overcome these drawbacks.

3.2.1 Short-Message Allgather Collective Designs

In the short message range, recursive doubling allgather[52] is popular in practice owing to its asymptotic complexity of $O(\log N)$ steps. Figure 3.2(a) shows the regular steps in orchestrating recursive doubling allgather on two Xeon-Phi equipped nodes, each consisting of two host processes and two MIC processes. At the end of each step, each process exchanges accumulated data with a new neighbor and in each step the size of the data doubles. At the end of $\log(N)$ steps each process eventually has data exposed by every other process.

As message size exponentially increases with step of the algorithm, the last step is the largest of transfers with a size of $N \times M$ ($N$ is the number of processes in the group and $M$...
is the initial size of message). As shown in Table 3.1, the communication link between two 
MIC processes residing on different nodes suffers from long latencies, which is marked as 
red arrows in the diagram. This being the case, depending on the process configuration, 
there is a likelihood that the size of data exchanged along such weak links may be the 
largest. This degrades the overall performance despite being asymptotically equivalent to 
the original algorithm.

To overcome such scenarios, we propose rescheduling the order of the recursive dou-
bling, as shown in Algorithm 1, which we refer to as RD-RESCHEDULE. First, weights 
are assigned to different neighbors based on the location of the process with which data is 
exchanged, as shown in Algorithm 1. Then, every process maintains a list of neighbors in 
reverse sorted weight order. For instance, if the process lies within the same platform (ei-
ther the same MIC or same host), the weight assigned to that exchange is 1. If the process 
resides within the same node but not in the same platform then the weight assigned is 2. 
If the process resides outside the node, then weight assigned is 3. The weights assigned 
reflects the cost of the links within and outside the node (Table 3.1). This weight assign-
ment and the subsequent exchange steps in the reverse order of weights naturally ensure 
that the least amount of data is exchanged on the slowest links. As a result, the largest of 
data transfers are orchestrated over the fastest links available. The weight assignment also 
ensures that in the majority of cases, the largest transfers are as local to the process as pos-
sible. It must be noted, however, that if the number of steps is large (as in the case of large 
MPI jobs) then a large number of exchanges still occur over weak links. Even though the 
priority_list_build_procedure differentiates three kinds of links as explained above for MIC 
clusters or similar setups, the procedure is extensible to differentiation based on processes
located within the same rack, or within the same cluster. Figure 3.2(b) gives a simple example of how the reordering applies on an 8-process allgather operation. The heaviest weight of remote MIC-to-MIC communication (step 3 in Figure 3.2(a)) now is re-assigned to step 1 in Figure 3.2(b) to match with the least data load. At the same time, the lightest weight of local MIC-to-MIC is re-assigned to step 3 with the most data load. Hence, the total latency reduces from \((T_{MML} + 2T_{HML} + 4T_{MMR})\Delta m\) to \((T_{MMR} + 2T_{HML} + 4T_{MML})\Delta m\), where \(\Delta m\) is the message length. To ensure that contiguous transfers are used even in the modified recursive doubling we use a temporary buffer for transfer steps and finally copy data back to user buffer on completion.

![Recursive doubling allgather algorithm](image)

(a) Default exchange

(b) Reordered exchange

Figure 3.2: Recursive doubling allgather algorithm
priority_list_build()
for (i = 0 to (log(N) - 1)) do
    if (dst[i] ∈ platform) then
        weights[i] = 1
    end
    else if (dst[i] ∈ compute_node) then
        weights[i] = 2
    end
    else
        weights[i] = 3
    end
end
sort_dst_by_weight = sort(dst, weights)
end
rescheduled_recursive_doubling_allgather()

tmp_buf = Allocatesize(comm_size * unit_message_size)
for (i = 0 to (log(N) - 1)) do
    dst = src = sorted_dst[i]
    MPI_Sendrecv()//toandfromsorted_dst[i]
end
copy_to_user_buf(tmp_buf, user_buf)
reordered_virtual_ring_allgather()
if (num_host_processes_in_node > 1) then
    Nh = get_num_hosts
    Nm = get_num_hosts
    if (is_mic_process) then
        virtual_rank = own_rank - (Nh - 1)
    end
    else
        if (own_rank % (Nh + Nm)! = 0) then
            virtual_rank = own_rank + Nm
        end
    end
left = (virtual_rank + comm_size - 1) % comm_size
left = index_of(left)
right = (virtual_rank + 1) % comm_size
right = index_of(right)
(run_regular_allgather_ring_with_adjusted_receive_and_send_offsets)
end

Algorithm 1: Building priority list, Weighted Recursive doubling and reordered Ring Allgather
3.2.2 Large-Message Allgather Collective Designs

The recursive-doubling algorithm is not preferred in large message or very large job allgather operations owing to latency involved in performing very large exchanges, especially when nearing the final steps of the algorithm. A popular algorithm that is used in the large message range is the ring-based allgather algorithm [52], which involves each process exchanging one message unit with its left and right neighbors for \( N - 1 \) rounds in a collective that involves \( N \) processes. In each step, each process receives the message unit from a unique source process through the left neighbor and passes along one message unit to its right neighbor. Figure 3.3(a) shows the original ring-based algorithm within two nodes, which include two host processes and two MIC processes respectively on each of the nodes.

The performance of the ring-based allgather algorithm is dictated by the latency of the slowest step in the ring. The slowest step in an allgather ring involving processes in a MIC cluster is the step involving transfer from one MIC process to another MIC process on a remote node, as marked red in Figure 3.3(a). This step is quite likely for a given ordering of MPI processes. The latency for the original ring-based allgather algorithm then can be represented as \( (N - 1)T_{MMR}\Delta m \). We can avoid this scenario by reordering the original ring, as shown in Algorithm 1, where we overlay a virtual ring that sandwiches the MIC processes between host processes within a node, as shown in Figure 3.3(b). We refer to this adaptation as RING-REORDER. This ensures that a MIC process never communicates directly with another MIC process on a remote node. In the subsequent reordered ring, the slowest step is the one between two MIC processes within the same node. The latency of reordered ring becomes \( (N - 1)T_{MML}\Delta m \). According to the point-to-point communication
study in Section 3.1, the relative latency of ring-based allgather algorithm can drop from 17 to 8.

![Diagram of ring-based allgather algorithm]

**Figure 3.3: Ring-based allgather algorithm**

Though reordered ring allgather (Algorithm 1) offers a way of switching the slowest step with a less expensive one, when the number of processes on the MIC increases, the bidirectional ring within a single MIC platform becomes the bottleneck. To avoid host processes from incurring this cost we propose the design as shown in Figure 3.4 where two rings are established - a host-ring and a mic-ring, referred to as **RING-DUAL**. The host ring consists of all host processes within the communicator and the mic-ring consists of just the MIC processes within a compute node. Next, the allgather proceeds with the two rings operating concurrently except that a host and a MIC process on the node indicate to each other through marker flags about the arrival of messages from different ranks and exchange of data between these two proceed asynchronously without intervention from one another through the DMA-based SCIF API [6]. This ensures that the bottleneck in the MIC bidirectional ring does not affect the faster ring (using IB) interconnecting the host MPI processes. While this approach reduces the average latency of the operation, it does induce skew owing to the host MPI processes finishing earlier than their MIC counterparts.
Figure 3.4: Dual-Ring-based Allgather
3.3 Heterogeneity-Aware Alltoall Designs

All-to-all is one of the densest collectives that is used in MPI programs and is a critical part of many linear algebra routines. State-of-the-art All-to-all algorithms suffer from repeated use of sub-optimal paths and here we propose optimizations necessary for the symmetric mode of operation on MIC clusters.

3.3.1 Short-Message All-to-All Collective Designs

Bruck’s All-to-all algorithm is $\log(N)$-step MPI operation that is used for small messages and it involves two phases [52]. The first phase consists of a local column reordering of elements and the second phase consists of a $\log(N)$-step inter-process communication with each step involving $\Delta m \frac{N}{2}$-sized data transfers, where $\Delta m$ represents the message length to be exchanged between every pair of processes. There is, however, a dependency that exists between step $i$ and step $i + 1$ of the $\log(N)$ steps. Hence, a delay in one step propagates across to subsequent steps. This is a highly likely scenario when MIC processes are involved, especially when there are exchanges between MIC process on two different nodes. In each step, the maximum latency can be one of $T_{MML}$, $T_{MHL}$, $T_{MHR}$ or $T_{MMR}$. As shown in Table 3.1, for small messages, the latency for accessing a local MIC memory is almost the same as accessing a remote MIC memory. To simplify the analysis, we use $T_{MMR}$ to represent the per message latency in a single step. Thus the total cost for the Bruck’s All-to-all algorithm can be presented as $T_{MMR}\Delta m \frac{N}{2} \log(N)$, where $\Delta m$ is the size of the message exchanged for an $N$-process collective. To avoid such delays we adopt two methods in view of architectural limitations that are imposed by the PCIe.
The first method involves bringing all data from MIC processes to host processes (pre-loading) in a round-robin mapped manner within a node, as shown in Algorithm 2, referred to as BRUCK-STAGED. Then, a second phase that involves the regular Bruck’s all-to-all algorithm is executed, with the host performing all the necessary operations for the MIC processes it is responsible for. On the completion of the second phase, the data corresponding to the MIC processes are sent back and a local column reordering takes place (post-loading). Through the pre-loading, the bottleneck cost $T_{MMR}$ can be replaced by $T_{HHR}$, at the cost of pre-loading overhead. The overall cost then is reduced to $T_{HHR} \Delta m \frac{N}{2} \log(N) + 2(T_{MHL} \Delta m N)$.

The drawback of this approach lies in the fact that all of the data that the MIC processes intend to exchange with other processes need to be brought in (in the pre-loading step) before the second phase begins and sent back (in post-loading) which stresses the PCIe bandwidth. This is especially expensive when the number of processes involved in the collective is large. Furthermore, this leads a phase where the inter-node network remains idle for the entire duration of pre-loading. When we reach such limitations, we adopt a more on-demand approach instead of staging all the data on host memory. The method is especially applicable to large message all-to-all algorithms such as pairwise that we discuss in the next subsection.

### 3.3.2 Large Message All-to-All

For large message all-to-all operations, the pairwise exchange algorithm is used where each process performs $N - 1$ steps and in each step, it exchanges data with another process involved in an $N$-process collective [52]. At a given step $i$, there are $N/2$ pairs of processes exchanging data with each other. Figure 3.5(a) illustrates the default scheme for a 4 process
host_mic_mapping()
if (node_rank == 0) then
| \(N_h = \text{get num host processes in node}\)
| \(N_m = \text{get num mic processes in node}\)
if \((N_h == N_m)\) then
| Establish \(1 \to 1\) mapping
else
| Establish round robin many \(-\to 1\) mapping
end
peers_list = generate_map_array
end

MPI_Scatter(peers_list, ..., node_communicator)
(root = 0)

staged_bruck_alltoall()
if (host_MIC_mapping_established) then
if (host_process) then
    for mic_process \(\in \text{dependent peers}\) do
      MPI_Irecv(&temp_buffer[mic_process],...
      (from mic_process)
    end
    MPI_Waitall(…)
    for \(i = 0 \ldots (\log(N) - 1)\) do
      perform_default_bruck_for_own_rank
      perform_default_bruck_for_dependent_peers
      (Local copies performed when dst rank’s data is local)
    end
    for mic_process \(\in \text{dependent peers}\) do
      MPI_Isend(&temp_buffer[mic_process],...
      (to mic_process)
    end
    MPI_Waitall(…)
else
    MPI_Send() (to host peer)
    MPI_Recv() (from host peer)
end
end
end

Algorithm 2: Host-MIC-process Mapping, Staged Bruck’s Alltoall
Algorithm 3: Selective Rerouted Alltoall

```plaintext
selective_rerouted_pairwise_alltoall()
if (host_MIC_mapping_established) then
  for i = 0...N - 1 do
    if host_process then
      dst = src = own_rank ∧ i
      MPI_Send() (to dst)
      MPI_Recv() (from src)
      for mic_process ∈ dependent_peers do
        peer_dst = peer_src = mic_process_rank ∧ i
        if peer_dst_outside_node then
          MPI_Recv() (from mic_process)
        end
      end
    while (outstanding_requests! = 0) do
      MPI_Waitany(...)
      if (received_message_from_mic_process) then
        MPI_Send() (to corresponding_peer_dst)
      end
    end
  end
else
  dst = src = own_rank ∧ i
  if (peer_dst_outside_node) then
    MPI_Send() (to host_peer)
    MPI_Recv() (to MPLANYSOURCE)
  end
  else
    MPI_Send() (to dst)
    MPI_Recv() (from src)
  end
  MPI_Wait() (Send request)
  MPI_Wait() (Recv request)
end
else
  run_default_pairwise_alltoall
end
```
pairwise all-to-all operation. It is important to note here that exchanges are blocking and each step $i$, involves a process completion of sending and receiving data to and from its partner in that step. Delays in such exchanges can propagate and affect subsequent steps through skews. For the large message range, we have observed that any communication involving a remote MIC (remote host-to-MIC, remote MIC-to-host, and remote MIC-to-MIC) has a relative latency as 17, which is much higher than any other communication links (marked as red in Figure 3.5(a)). Hence it is necessary to avoid such delays when possible. All-to-all operations involving MIC processes are bound to run into such situations. One possible way to avoid this is to defer such costly steps to final or near final stages similar to assigning weights in the allgather as in Section 3.2. This however helps improve the average latency experienced by all processes alone. The other way to not incur such delays is for the MIC processes to transfer all data to hosts and let the hosts orchestrate the entire operation and before finally receiving resultant data. As mentioned before though, the pair-wise all-to-all is used for large messages and transfer large quantities of data to hosts.
through the PCIe stresses the link which has a limited bandwidth. This approach also adds to host processes’ memory overheads. Hence we propose to re-route all transfers which have MIC source and a destination outside the node for each step of the algorithm 3. Figure 3.5(b) indicates the re-routing scheme and is referred to as PAIRWISE-SLR. The green arrows in the diagram show the re-routing path. The implication is that at a given step the host needs to reserve a single buffer of the size of message being exchanged for each MIC process that it is responsible for. Furthermore, the effective exchange time between a MIC process and any other process on a remote node reduces considerably as the costly read operations from MIC memory by the NIC are avoided. Through this re-routing, the bottleneck cost of $T_{MMR}$ and $T_{MMH}$ are transformed to $2T_{MHL} + T_{HHR} + \alpha$ and $T_{MHL} + T_{HHR} + \alpha$, respectively. $\alpha$ represents the overhead due to sequential processing of host and MIC requests. As alluded to earlier, this very approach can be used to realize Bruck’s all-to-all algorithm when message sizes get too large that not all can be transferred to hosts concurrently and this method is referred to as BRUCK-SLR.

### 3.4 Performance Evaluation

Here, we discuss the details of experimental setup and present the micro-benchmark results of the proposed All-to-all and Allgather collectives and provide an analysis of the observed results. Then we compare the benefits of the proposed approach with Intel MPI library and show the impact of designs with the P3DFFT application which is largely influenced by the performance of All-to-All operation.

**Experimental Setup** The experiments were conducted using up to 32 Sandy Bridge-EP nodes on Stampede cluster at TACC. The nodes comprised of dual Intel Xeon E5-2680 2.66GHz eight-core processors with 32 GB of main memory. Further, the compute nodes
featured an Intel Xeon Phi coprocessor (Knight’s Corner). The PCI-E device features 61 cores, with an aggregate of 244 hardware threads. Despite the surplus of compute resources available on the MIC, the device has a limited memory of 8 GB. Mellanox OpenFabrics Enterprise Edition InfiniBand stack was used with Mellanox HCAs for reading and writing to both the host and the Xeon Phi memories. The compute nodes were connected to Mellanox FDR switch SX6036. Our designs and evaluations were based on the MVAPICH2-2.0 release and we used the OSU Micro-Benchmark suite for our analysis. We report the average of the All-to-all collective latency results that are averaged across multiple runs of the benchmark and concentrate on message regions most commonly used in applications.

3.4.1 Micro-benchmark Evaluation

The OSU Micro-benchmark (OMB) suite runs collective operations for either 1,000 or 100 iterations depending on the size of the message unit being exchanged. We use the following notation to indicate the different designs proposed in this chapter.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRUCK-STAGED</td>
<td>Bruck’s Alltoall where MIC data is staged on the host</td>
</tr>
<tr>
<td>BRUCK-SLR</td>
<td>Bruck’s Alltoall with selective rerouting</td>
</tr>
<tr>
<td>PAIRWISE-SLR</td>
<td>Pair-wise Alltoall with selective rerouting</td>
</tr>
<tr>
<td>RD-RESCHEDULE</td>
<td>Recursive doubling Allgather with exchange ordering rescheduled</td>
</tr>
<tr>
<td>RING-REORDER</td>
<td>Ring Allgather with reordered virtual ring</td>
</tr>
<tr>
<td>RING-DUAL</td>
<td>Dual ring Allgather</td>
</tr>
</tbody>
</table>

Table 3.2: Notation used to identify different collectives

Dense collectives such as the allgather and alltoall have a bigger memory footprint compared to most other collectives. For instance, for the MPI_Alltoall collective each process
allocates $2 \times M \times N$ amount of buffer at the start of the collective, where $M$ is the maximum size of message unit being transferred and $N$ is the number of processes involved in the collective. There are $2 \times M \times N$ units of data used because each process allocates a personal message for every other process in the collective and also receives one message unit from every other process. Due to the limited amount of memory (8GB) running on the MIC, we restrict our study to using either 16 or 32 nodes with a maximum of 16 processes on either the host or the MIC platform. Also, we believe that, given the significant amount of threads available on the MIC, a more suitable model that fits symmetric mode of operation on MIC clusters is that of using a limited number of MPI processes on the MIC with each process spawning multiple OpenMP threads. For this reason, as we discuss in the application evaluation we choose P3DFFT to show benefits of the proposed design.

**Allgather Results** The original recursive doubling allgather is suitable for small message ranges. The benefit of RD-RESCHEDULE is mostly in the message of 64 bytes - 1 Kilobyte sized messages with about 50-60% reduction in latency. Beyond this point, as mentioned in Section 3.2, the copy overheads from the intermediate temporary to user buffer begin to outweigh the benefits of sending smaller sized messages on lesser performing links. The benefits range in 50-75% for the 16 and 32 node case shown in Figure 3.6(a) and Figure 3.6(c).

In the large message range, RING-REORDER and RING-DUAL aim to overcome the bottlenecks of the traditional ring allgather approach. Figure 3.7(a) shows the improvement obtained for a 4 node case with 2 processes on each of the MIC and the host platform. We observe improvements in the large message range in order of 50-60% beyond 512KB. However, we noted that with the increase in the number of MIC processes running on the Xeon Phi, the improvements become negligible or there was degradation with the reordered
Figure 3.6: Performance of Small Message MPI Allgather

(a) 16 Nodes: 16 Procs/Host and 16 Procs/MIC  
(b) 32 Nodes: 8 Procs/Host and 8 Procs/MIC  
(c) 32 Nodes: 16 Procs/Host and 16 Procs/MIC
ring. We attribute this degradation in latency of message transfers on the Xeon Phi to the limitations of the bi-directional ring on the MIC in the presence of increasing number of processes and the fact that two MIC processes are involved in host-to-MIC or MIC-to-host transfers. The RING-DUAL scheme, on the other hand, alleviates this owing to a single process being responsible for data transfers into and out of the MIC. Furthermore, the overlapped asynchronous transfers ensure that the hosts are not slowed down by the slowness induced in the presence of a large number of MIC processes in the Xeon Phi. The improvements from the dual ring design are evident in Figure 3.6(c) and up to 79% improvement is seen for 32 node setup with 8 processes on each of the platforms in the large message range.

**Alltoall Results** Bruck’s alltoall algorithm is suitable for small messages as the amount data exchanged at each of the $\log(N)$ steps amount to $N \times M/2$, where $M$ is the maximum size of message unit being transferred and $N$ is the number of processes involved in the collective. We observe that BRUCK-STAGED is most applicable for message ranges between 1byte to 64 bytes for 16 nodes. Beyond 64 bytes the copy overheads of bringing all of the MIC data to host peer processes can start choking the PCI channel and cause a huge increase in latency as seen in Figure 3.8(c). At this point, we note that BRUCK-SLR, similar in design to PAIRWISE-SLR, alleviates the large copy overheads.

In the large message range Figure 3.9(a) and Figure 3.9(b) show that PAIRWISE-SLR yields reduction in latency by up to 60% for the 16-node case.

In Figure 3.10(a), Figure 3.10(b), Figure 3.11(a) and Figure 3.11(b) a comparison with Intel MPI library is provided. The message regions, where we target optimizations, benefits from avoiding weak communication links and hence has a reduced latency compared to the default MVAPICH2-2.0 library and Intel library.
Figure 3.7: Performance of Large Message MPI_Allgather

(a) 4 Nodes: 4 Procs/Host and 4 Procs/MIC  
(b) 16 Nodes: 8 Procs/Host and 8 Procs/MIC  
(c) 32 Nodes: 8 Procs/Host and 8 Procs/MIC
Figure 3.8: Performance of Small Message MPI_Alltoall on 16 Xeon Phi Nodes
Figure 3.9: Performance of Large Message MPI_Alltoall on 16 Nodes

(a) 4 Procs/Host and 4 Procs/MIC
(b) 8 Procs/Host and 8 Procs/MIC

Figure 3.10: Overall Performance Comparison of MPI_Alltoall, 16 Nodes: 8 Procs/Host and 8 Procs/MIC

(a) Small Messages
(b) Large Messages
3.4.2 Evaluation with an Application Kernel

We use the P3DFFT application kernel to further evaluate the impact of proposed designs. P3DFFT is a popular library for parallel three-dimensional Fast Fourier Transforms. It is written using Fortran and MPI+OpenMP hybrid model. The version considered in 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. Figure 3.12(a) and Figure 3.12(b) show the time spent in communication, which is composed of several All-to-all operations, and the reductions in communication time from using the proposed designs. For the smaller sized problem, BRUCK-STAGED gets used and PAIRWISE-SLR for the larger variant of the problem. Figures 3.12(a) and 3.12(b) contrast the execution time of using default and optimized Alltoall algorithms with 512 processes for problem sizes 512x512x512 and 2Kx2Kx2K respectively and an improvement of up to 38% is seen.
3.5 Related Work

Researchers have explored various ways of programming applications to utilize Intel MIC’s abundant compute resources. 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 OpenMP-based programmability aspects [29]. Likewise, Deisher et al. have proposed a way to accelerate LU factorization by offloading computations on to the MIC achieving as much 80% efficiency [37]. Rosales [12] determined that writing a hybrid code that worked in symmetric execution mode was more effective in terms of development time and future scalability.

MPI-based programming is a more natural fit for large scale applications. Larry Meadows’s work investigates the performance of WRF on the Xeon Phi using symmetric mode of computation and shows significant speedup [39]. MPICH-3.0.3, an open-source implementation of MPI, also has support for intra-MIC communication using shared memory, TCP/IP, and SCIF-based communication [40].
Optimizing and tuning collective operation on homogeneous host-based clusters has been well studied. Ramos et al., have explored the challenges in modeling communication in cache-coherent SMP systems [57]. The performance of All-to-All and All-Gather MPI operations, in particular, has been of interest to application and algorithm developers alike. These operations are two of the most intense collective operations used in several HPC applications and kernels. Kumar et al. proposed a non-leader based scheme to optimize All-to-All collective for multi-core systems. Kandalla et al. proposed a multi-leader based algorithm for MPI Allgather. Singh et al. exploited a dynamic-staging mechanism to optimize All-to-All operations that involve GPUs.

In our prior work, we studied the Gather, Bcast, and reduction operations, and its intrinsic characteristics on the MIC architecture [25]. Our solution was able to alleviate the inherent bottlenecks using hierarchical, leader-based techniques and overlapping methods. These techniques are not directly applicable to the dense collectives studied in this chapter owing to the sheer volume of data exchanged in the MPI Alltoall and MPI Allgather collectives. In this chapter, we propose adaptations to some of the classic algorithms used to implement the MPI Alltoall and MPI Allgather and a few optimizations to make these operations scalable across several nodes that are provisioned with the Xeon Phi cards. Kielmann et al. have proposed optimizations to collectives for geographically separated systems (including the Allgather operation) but their work hinges on ‘co-ordinator’ processes to overcome bottlenecks [27]. Our techniques relies on adaptations to certain popular algorithms that make them suitable for heterogeneous environments like MIC clusters.
3.6 Summary

Based on our evaluations and analyses, it is evident that the Xeon Phi architecture has inherent limitations. Although existing collective algorithms are still relevant on this architecture, they are not fully efficient. We have studied the bottlenecks incurred on the state-of-art collective algorithms, for MPI_Allgather and MPI_Alltoall operations. Our evaluations clearly identify the ideal data-paths in heterogeneous architectures provisioned with MICs. Based on the findings, we proposed enhanced algorithms for these dense collectives, which addresses the asymmetry present in the network paths. Finally, we evaluated our algorithms with the most prevalent MPI benchmarks, and also with a real-world application kernel. For the Allgather operation, we see up to 75% improvement in small-message latencies, and up to 79% for large messages. In case of the All-to-All operation, we see up to 50%-60% improvement for short and large messages. With the P3DFFT application kernel, we see 38% reduction in the total runtime when our enhanced dense-collective algorithms are used.
Chapter 4: Leveraging Novel Network Capabilities for High Performance GPU Collectives

In this chapter, we discuss the inefficiencies in using traditional host-staging approach for frequently used GPU broadcast operations (using multicast) in throughput-critical streaming applications. We then describe, how novel network mechanisms such as GPUDirect RDMA (GDR) can be leveraged with traditional InfiniBand capabilities such as scatter-gather-list abstractions and hardware multicast to design efficient multicast-based broadcast operations. Next, we show how mechanisms such as GDR can be combined with existing network offload techniques to propose non-blocking GPU collectives. We show how architectural limitations can be overcome by using function invocation techniques such as CUDA callback mechanisms. Experimental evaluation shows that the proposed designs provide significant improvements in latency and throughput for the broadcast operation and near-perfect overlap for dense non-blocking collectives from GPU buffers.

4.1 Challenges in Multicasting GPU Data and Design Alternatives

Figure 4.1(a) shows the benefits of using multicast capability at scale to realize broadcast operation but Figure 4.1(b) shows the drawbacks in staging GPU data for similar GPU data broadcast operations using the scheme shown in Figure 4.2. As mentioned in Section 2.1.3, multicast can be performed using UD QPs alone on IB networks. For this work,
(a) Host-memory buffer broadcast with and without Multicast

(b) Host-memory Multicast and Host-staged GPU-memory Multicast Comparison

Figure 4.1: Use of Multicast for Broadcast operations

Figure 4.2: Host-staged GPU-memory Multicast (Naive Scheme)
the traditional and state-of-the-art NACK-based multicast scheme is used and we propose methods to avoid drawbacks of host staging. There are two main mechanisms for orchestrating multicasts on high performance networks based on registration costs:

4.1.1 Using an Eager Protocol

In an eager-mode of operation, a set of ‘eager’ buffers are pre-allocated and pinned (locked memory) with the HCA of the node at both the source and the sink communicating processes. When a multicast is to be performed, the source of the broadcast operation copies the actual user data to be broadcasted into the head-of-the-line list of eager buffers and issues a multicast operation on that buffer. Multiple sinks, awaiting the arrival of this segment (no greater than 2 Kilobytes owing to an MTU restriction), are then notified on arrival and a subsequent data transfer occurs between the sink-side eager buffer to the actual user buffer.

There are two possible design alternatives that can be used to realize the protocol for GPU buffers. Firstly, the eager buffers can be allocated on either the host (as traditionally done) or directly on the GPU memory. These eager buffers are then registered with IB HCA even if the memory being registered resides on the GPU (with GDR support). Once registered, a multicast-send or a multicast-receive request can specify the addresses of these eager buffers from/to where data is to be sent/received. However, there are drawbacks with both these approaches. When eager buffers reside on the host, a call to cudaMemcpy from GPU device to host memory is required to copy the contents of the user exposed GPU buffer to the eager buffers and vice versa at the receiver once the broadcast data has arrived. When eager buffers reside on the GPU, a call to cudaMemcpy for an intra-GPU transfer is required to copy the contents of the user exposed GPU buffer to the eager buffers. Both
of these methods require an expensive `cudaMemcpy` which involves requesting the `cuda-runtime` to perform actions on behalf of the host and hence not very viable. Figure 4.3 depicts the eager scheme for multicasting GPU data.

![Diagram of GPU-memory Multicast (Eager Protocol)](image)

**Figure 4.3:** GPU-memory Multicast (Eager Protocol)

### 4.1.2 Using a Rendezvous Protocol

An alternative to the above protocol is to directly register user exposed GPU buffers on the fly and then perform the `multicast-send` and `multicast-recv` calls. On the face of it, this alternative would avoid the expensive `cudaMemcpy` calls but tends to be prohibitively expensive for short messages owing to the expensive memory registration cost. Furthermore, there is a need for the prototypical handshake between the receivers and the source (using a gather operation) before the multicast operation can be called by the source of the broadcast. For larger messages, as UD transport allows a maximum of 2 Kilobytes of data to be transferred in a single transfer, explicit packetization is required and hence ordering can
prove problematic as buffers are directly touched and handling out-of-order packets will require the use of either `cudaMemcpy` call or require another multicast operation. Figure 4.4 depicts the rendezvous scheme for multicasting GPU data.

![Diagram of GPU-memory Multicast (Rendezvous Protocol)](image)

**Figure 4.4: GPU-memory Multicast (Rendezvous Protocol)**

### 4.1.3 Proposed Design - GDR-SGL-MCAST

The proposed design chooses a hybrid approach where the eager buffers are maintained in the host memory for control information exchange. At the receiving end, persistent and registered buffers are used to avoid the expensive registration call in the critical path of the multicast phase. The use of persistent registered buffers, however, is applicable primarily in the context of streaming applications owing to real-time nature of applying
certain transformations on arriving data from the same memory location. The details of the proposed design are as follows:

**Use of Scatter-Gather-List** The primary challenge in orchestrating the multicast is getting both multicast control information (which holds packet sequence number, multicast group id, etc) and GPU data into the same packet. The challenge lies in not being able to generate multicast control information directly on the GPU nor being able to efficiently transfer control information directly to eager packets on the GPU’s memory. If this was possible, then GPU data multicast is trivial as both control information encoding and decoding logic and data packing can be carried out solely in the GPU memory and then multicast could be issued directly using GDR.

To avoid the expensive `cudaMemcpy`, the proposed design uses the *scatter-gather-list* (SGL) abstraction that the IB architecture provides. Through this abstraction, the sender/receiver sides can specify an array of ‘scatter-gather’ elements. Each element of the array consists of an ‘address’ and a ‘length’ argument. Hence at the sender side, all elements specified are *gathered* from specified addresses and in specified lengths by the IB HCA without interrupting the host at any point. Through GDR, one can specify a GPU address in the scatter-gather list as well. On the receiver side, as an inverse operation, the incoming data is *scattered* into destinations specified in the scatter-gather list in specified amounts. Figure 4.5 shows the schematic diagram for (SGL) at the source and sinks.

**Fast-GPU-Multicast Methodology** While scatter-gather-list (SGL) is a useful abstraction to avoid expensive paths, there is a dependence associated with the use of SGL — the source cannot start multicasting before all the recipients of these packets have arrived. For instance, if the root of the broadcast operation was to multicast GPU data before one of the sinks have arrived, then due to lack of non-specification of a GPU memory address in
the receiver’s scatter list, there is incorrect data transmission (the packet may be silently
dropped). This dependence can be satisfied by allowing the source to know when all the
other sinks have arrived in the *handshake* phase. We achieve this using a 0-byte gather (us-
ing a k-nomial implementation). For the subsequent series of multicasts, we use a persistent
GPU buffers in tandem with existing UD eager buffers on the host memory to encapsulate
multicast control information (we view the use of persistent buffers to be fair in the con-
text of streaming applications). Hence, for all ensuing transfers, the SGL consists of 2
entries: 1. an address pointing to the host eager UD buffer which holds related control
information (with length amounting to control information size) and 2. an address pointing
to the GPU address from which data is to be broadcasted. While doing so it is ensured
that the combined size of the GPU buffer and the multicast control information is less than
or equal the UD MTU else packetization is used. Figure 4.6 shows the orchestration of
GDR-SGL-MCAST scheme at the source and sinks.

![Diagram]

Figure 4.5: Scatter-gather list representation for MCAST packets
Handling reordering As UD transport does not offer reliability and does not make ordering guarantees, this needs to be explicitly handled. Even though IB networks see a very small fraction of packet drops, the proposed design piggybacks on sliding-window based error handling mechanisms to ensure that applications see no errors in transmissions. Traditional NACK-based protocol is used in case receivers/sinks do not see a packet sequence number within an empirically derived timeout duration. Within this duration, if the source does receive a reliable NACK then multicast is repeated and all but those sinks who did not receive the packet originally drop the repeated packet. In other words, sinks which successfully received the packet in the first attempt do not progress their queue heads.

There is small probability of packet re-ordering occurring on IB networks as well. For multicasts of message size less than the MTU, there are no ordering concerns as multicast occurs with a single operation. For message sizes greater than the MTU, packetization is a
requirement and to deal with reordering the following approach is used. 1) The reordered packet is received in the original GPU addresses; 2) If on completion of message multicast, reordering is detected then, a temporary buffer space within the GPU is used to shuffle the data into original positions using *Device-to-device - cudaMemcpy*; and 3) If burst re-ordering is detected then cost of *cudaMemcpy* is amortized by shuffling en masse. This procedure ensures that shuffling does not come in the multicast critical path and deferred to a post-processing step. The low-probability of reordering coupled with handling in the non-critical path allows for better tolerance of re-ordering even though it uses the expensive *cudaMemcpy* calls.

**Discussion on MPI_Bcast** The use of the proposed design for MPI_Bcast operation on GPU buffers is limited by the use of the zero-byte gather (for each MPI_Bcast operation). While the latency of the GDR-SGL-MCAST operation for a given message size remains constant or increases by a very small amount, the zero-byte gather grows logarithmically with system size and hence limited in terms of scalability.

### 4.2 Design Considerations for Dense GPU Non-blocking Collectives (NBCs) and Proposed Approaches

Blocking MPI collectives such as *Alltoall, Allgather, Scatter, Gather* and their v-variants can be very expensive operations especially when the source and target buffers of these operations are on GPU memories, especially at scale. Thus it is profitable for the application to place such dense communication operations for progression in the background while the majority of the cycles are spent on progressing computation when possible. This motivates the need for efficient non-blocking collective operations that can offer good overlap at a
latency comparable to that offered in the blocking context. Here, we discuss the different mechanisms for realizing non-blocking collective operations from/to GPU buffers and propose designs for achieving good overlap at competitive latency for these transfers. We also describe how to make the designs portable to the majority of non-blocking collective algorithms that MPICH and its derivatives such as MVAPICH have to offer.

4.2.1 Progression Through Host-assistance

Most MPI implementations support collective communication operations between GPU memories using a host-staged design that involves data movement from 1) source buffer to intermediate host memory ($D2H$); 2) a network transfer to intermediate host memory at remote destination ($H2H$); and 3) a final transfer to the destination GPU buffer ($H2D$). Depending on message size, the three steps are executed once or for each segment of a large message, in a pipelined manner. On GPU-enabled InfiniBand clusters, this can be realized using the `cudaMemcpy` and its variants for stages 1 and 3, and using `verbs` operations for stage 2. This fundamental scheme is used with rudimentary point-to-point operations allowing non-blocking collective operations composed of `Isend/Irecv/Wait/Waitall` operations to be designed. The reliance on this pipeline to realize non-blocking collectives, however, places the burden of transfer entirely on the host processor (Figure 4.7). Hence, while this scheme offers good latency and bandwidth for blocking collectives, it offers very little overlap and not a good candidate for NBC with overlap requirements.

4.2.2 Progression Through Host-assistance and Offload Mechanisms

For NBCs that prioritize overlap, it is necessary to forgo the use of the `pipeline` channel to reduce the active participation of the host processor. With this restriction, a naive design to take advantage of the offload mechanisms is to transfer all of the GPU content to an
intermediate host buffer en masse and exploit the HCA’s asynchronous processing capabilities to completely offload message processing and relieve the host processor of network responsibilities from that point onwards. This can be realized through the use of CORE-Direct Offload Engine from Mellanox, making use of their experimental verbs API. The approach is to compose verbs instructions (referred to as a tasklist) to transfer data to intermediate host memories at remote destinations from the intermediate buffers at the source node. Naturally, this requires a transfer back to user-provided destination GPU buffer using a cudaMemcpy* in the corresponding MPI_Wait call at the remote destination. These stages are shown in Figure 4.8. This approach combats excessive involvement of the host processor at each step as in the pipelined approach. The primary drawback of this scheme, however, is that for operations such as MPI_Ialltoall, the preliminary transfer to host and the optional transfer back from intermediate host memory to destination GPU memory can be expensive. As these copies are in the critical path, they impact negatively on the both overlap and latency of NBCs.

4.2.3 Progression Through Offload Mechanisms and GDR

As indicated in Section 2.2.2, GDR technology allows for peer-to-peer communication between the HCA and GPU device within the node. Through this, GDR allows for the specification of GPU buffers in verbs instructions and transfers from/to GPUs are seamlessly carried out without requiring the intervention of the host memory. Leveraging both GDR and CORE-Direct offload mechanisms, here we propose a novel scheme that directly issues offload tasks that specify GPU memories (Figure 4.9). This scheme completely avoids the use of the PCI channel between the host and the GPU within the node. Thus, the host is responsible primarily for the initiation of offloaded network operations known
as the *tasklist*. This list is composed of tasks such as sending to a specific destination, receiving from a specific source, waiting for the reception of a message on specific completion queue, etc. Once the *tasklist* is populated and issued, the host is freed up to carry on with other compute/scheduling activities. Despite the seeming simplicity of the approach, this scheme suffers from the well-known peer-to-peer bandwidth limitation described in Section 2.3. Primarily, operations that involve the HCA reading large quantities of data from GPU memory and certain message ranges that involve the HCA writing into the GPU memory are known to have high latency and bandwidth penalties. Thus, this scheme can only be targeted for small and medium message sizes or if overlap alone is the primary goal and latency degradation can be tolerated.

### 4.2.4 Progression Through Offload mechanisms, GDR and CUDA Callback Mechanisms

CUDA offers a callback functionality feature that allows application developers to execute host code when a device operation completes in an execution stream called a CUDA stream. For instance, certain host code can be completed upon the completion of an asynchronous memory transfer operation realized through `cudaMemcpyAsync`. We leverage this callback functionality to overcome the PCIe limitations mentioned earlier. The idea we propose is as follows. The first step involves an asynchronous memory transfer operation from the device to an intermediate host memory location on a specific CUDA stream. This avoids the peer-to-peer data path from GPU to HCA which suffers from low bandwidth at medium and large message range. The next step involves adding a callback to the very same CUDA stream. This callback then invokes a function that issues a tasklist of network operations dependent on the completion of the asynchronous transfer from device to host memory. This callback mechanism ensures data dependence satisfaction criterion.
At the same time, it does not require host to poll on the completion of the asynchronous memory transfer from the GPU. The callback mechanism appears to initiate a lightweight thread that wakes up, executes the CPU code and returns to a sleep state. Once this is done, there are two options that fit different scenarios. One option is to have destination processes specify an intermediate host memory and then transfer data back to the GPU in MPI_Wait. The other option is to instead specify user GPU buffers directly and have data transferred directly to the device. When the write data path from HCA to GPU is not sub-optimal, this alternative serves well and avoids an additional cudaMemcpy operation. This scheme thus allows for good overlap as well as good latency for large message sizes (Figure 4.10).

4.2.5 Generalization Through Schedule Abstraction Reuse

Previous approaches which have leveraged on offload mechanisms through CORE-Direct have resorted to manually constructing their algorithms that are efficient due to low-level implementation but are not general and hence the process of port new designs to the plethora of collective algorithms is an arduous process. In this work, we map CORE-Direct’s tasklist abstraction to the schedule abstraction that is used by MPICH and its derivatives. The schedule abstraction primarily replaces all the send, receive, and local_copy operations that appear in a collective algorithm, with a corresponding schedule_send, schedule_recv and schedule_copy. In addition, the schedule can also consist of schedule_barrier which enforces completion of all previously scheduled operations before further processing can proceed. Thus any collective algorithm is a specification of a list of schedule_(send/recv/copy/bARRIER) operation. In strict terms, there are other schedule operations, but for the purposes of this work we map the CORE-Direct tasklist to this list of scheduled operations. The mapping involves using CORE-Direct’s task_send, task_recv,
\{\text{task\_send + task\_recv}\} (loopback) and \text{task\_wait} operations to realize their schedule counterparts. Each of these task generations requires specification of buffers, their memory registration handles, their notification completion queues, and other such handles. Adendum such as buffer registration functions and data structures, to capture resources related to the communicator and device, are used to assist the mapping. MPICH’s schedule processing method involves individually issuing MPI\_Isend/MPI\_Irecv to initiate schedule\_send/schedule\_recv and stopping at schedule\_barrier to wait for completion of all issued operations. Thus the host is largely involved in the progression of the non-blocking collective specified as a schedule. In contrast, using CORE-Direct’s tasklist, all tasks are completely specified and issued en masse and the HCA handles dependence satisfactions. Thus, this mapping makes the offload capability applicable to majority of the collective algorithms used by MPICH and its derivatives.

Figure 4.7: Naive GPU Non-Blocking Scatter
Figure 4.8: Staged-Offload GPU Non-Blocking Scatter

4.3 Performance Evaluation

Experimental setup

Experiments were conducted on Wilkes cluster at the University of Cambridge. The cluster comprised of 12-core Ivy Bridge Intel(R) Xeon(R) E5-2630 compute nodes running at 2.60 GHz with 64 GB RAM. The nodes were equipped with FDR ConnectX2 HCAs and NVIDIA K20c GPUs. The operating system used was Scientific Linux release 6.5 release (kernel version 2.6.32) and Mellanox OFED version MLNX_OFED_LINUX-2.1-1.0.6 which supports GPUDirect-RDMA (GDR). We used MVAPICH2-GDR [54] for comparison with the proposed designs.

Notation and Experimental Method
Figure 4.9: Offload-GDR GPU Non-Blocking Scatter
Figure 4.10: GDR-Offload-Callback GPU Non-Blocking Scatter

For multicast-based broadcast experiments, for the figures in this section, the legend ‘N-Nodes’ represents the test case of broadcast running on ‘N’ nodes with one process per node and with the broadcast data residing in the GPU memory. We also use the notation ‘HSM’ to indicate Host-staged-multicast and ‘GSM’ for the proposed GDR-SGL-MCAST design where applicable.

For GPU-based nonblocking experiments, Intel Micro-benchmark (IMB) suite version 4.0 is used for most of the evaluation. IMB’s micro-benchmarks measure overlap by first measuring the time $T$ for a combination of the non-blocking collective launch call and an immediate MPI_Wait call. This time $T$ is then used in a subsequent sequence composed of the launching of the non-blocking collective, a compute phase that runs for time $T$ and an MPI_Wait call. The overall time for this second sequence is used to measure the overlap potential of the collective. Hence sub-optimal overlap is an indication of the requirement
of CPU intervention to progress the collective. Furthermore, as our work proposes non-blocking collectives for GPU buffers, the overlap capability when kernels are executed on the GPU is of interest. Hence, a modified version of the benchmark that runs a GPU workload, that does a simple AXPY, alongside a CPU compute workload is used. Further, we use the following shorthand for overlap experiments:

- **Offload-Staged**: corresponds to the design introduced in Section 4.2.2. This design is used as our baseline.

- **Offload-GDR**: refers to the Progression through offload mechanisms and GDR design presented in Section 4.2.3.

- **Offload-Callback**: which is the Progression through offload mechanisms, GDR and CUDA call-back based design presented in Section 4.2.4.

As NBC operations are mainly target large message size to offer a potential for overlap, in our evaluation, we focus on medium to large message sizes. We present a use case for an all-to-all, one-to-all and all-to-one communication pattern using MPI_Ialltoall, MPI_Allgather, MPI_Is scatter, and MPI_Igather. The collectives are realized through the use of pairwise-alltoall, ring-allgather, direct/flat-scatter and direct/flat-gather respectively owing to extensive use of the algorithms for medium and large message range messages. For these algorithms, we present an evaluation with 16, 32 and 64 GPU-nodes.

### 4.3.1 Latency of multicast-based broadcast operation

In this section, we present the latencies of performing the broadcast operation using the proposed design GDR-SGL-MCAST and compare it with host-staged multicast of GPU buffers using MVAPICH2-GPU. Of the designs discussed in Section 4.1, the host-staged
approach proves the best in terms of scalability and hence we consider it as the base case for experiments to compare with the proposed design. The latency of sending ‘m-byte’ messages on a persistent buffer is measured as is typically the case in streaming applications.

The performance of the multicast latency for small messages seen with the Host-Staged approach is depicted in Figure 4.11(a). We can see that within the MTU range of 2 Kilobytes, the latency is nearly constant for different job sizes. However, we also observe that the least observed latency is a little over 20 microseconds. This is due to the fact that either device-to-host memory or host-to-device memory takes roughly 7-8 microseconds each way and the rest of the time is spent in the multicast. However, as seen in Figure 4.11(b), for the proposed GDR-SGL-MCAST, we see that within the MTU range of 2 Kilobytes, the latency is still nearly constant for different job sizes and the least observed is less than 10 microseconds. Device-to-host memory or host-to-device memory copy overheads are avoided through the use of scatter-gather lists and reduction in latency by up to 60% is seen for the short message sizes for a 64-node job.

Figures 4.12(a) and 4.12(b) depict the performance of medium message sized broadcast using the host-staged approach and the proposed GDR-SGL-MCAST approach respectively. As we can see, the sequential transfer of packets caused due to the packetization happening at the MTU of 2 Kilobytes takes its toll and hence we do not see the near flat latency seen in the short message range.

Figure 4.13(a) and Figure 4.13(b) contrast the latencies of the designs for different system sizes where GDR-SGL-MCAST outperforms the host-staged design. In Figure 4.14(a) and Figure 4.14(b), however, we see that the host-staged design outperforms GDR-MCAST-SGL for the majority of the range. This is due to the known PCIe peer-to-peer bottleneck
Figure 4.11: Performance comparison of short message multicast latency for GPU data

Figure 4.12: Performance comparison of medium message multicast latency for GPU data
that exists between the GPU and the IB HCA on Sandybridge and Ivybridge architectures in the medium and large message range. It is expected that this bottleneck will get addressed in future systems and for current systems, the proposed designs are not applicable to medium and large message regions.

Figure 4.13: Latency Comparison (small-sized messages)

4.3.2 Throughput of broadcast operation

In this section, we present the execution time of a synthetic benchmark composed of performing 1,000 back-to-back broadcast operations using the proposed design GDR-SGL-MCAST and compare it with host-staged multicast of GPU buffers using MVAPICH2-GPU. The dissemination phase of streaming applications where live data is constantly fed to sinks (where computations happen in a pipelined manner) is one such instance where the pattern used in the synthetic benchmark is exhibited.
Figures 4.15(a) and 4.15(b) show the execution time of the synthetic benchmark for the host-based multicast and GDR-SGL-MCAST designs (for different message sizes between 2-bytes and 512-bytes for different number of nodes). Even though the latency of the operation for the host-based multicast scheme is nearly constant as seen in Figure 4.11(a), the differences in execution time for the throughput benchmark is more pronounced. Conversely, with GDR-SGL-MCAST design we continue to see invariance in execution time as the scale of the job increases (In Figure 4.15(b), the jump in cost at 8-byte mark is anomalous but reproducible). Figure 4.15(c) contrasts the performance of the two designs and the proposed design shows a reduction in execution time by 3X-4X.

4.3.3 Latency of MPI broadcast

Lastly, we briefly discuss on the performance of MPI_Bcast (designed by combining proposed GDR-SGL-MCAST and a zero-byte-knomial-gather). As mentioned in the design section, even though the multicast performance remains constant, the zero-byte-gather
Figure 4.15: Execution time of throughput benchmark
steadily increases latency and as result the performance of GPU buffer MPI_Bcast shows a corresponding growth as shown in Figure 4.16(a). In Figure 4.16(b), the latency of segmented-multicast outweighs that of the zero-byte-gather and hence the difference is not stark.

![Graphs showing latency of MPI_Bcast using GPU data]

(a) Small-sized Messages  
(b) Medium-sized Messages

Figure 4.16: Latency of MPI_Bcast using GPU data

### 4.3.4 Overlap performance of MPI_Ialltoall operation

In this section, we present the overlap achieved by an MPI_Ialltoall operation using the proposed designs and compare it with host-staged scheme using GPU buffers. As depicted by Figure 4.17, on the three different system scales, the overlap of host-based staging design decreases with the increase of message size. This is mainly due to the overhead of the data copies. The overhead increases with message size and hence there is
a proportionate decrease in overlap potential. Similarly, on all system scales, the *Offload-GDR* and *Offload-Callback* show at least 80% overlap. While the performance of *Offload-Callback* is decreasing with message size due to the overhead of the copy, the *Offload-GDR* achieve a near perfect overlap with 99% computation/communication overlap on 64 GPU nodes with 1MB message size as shown in Figure 4.17(c). This effect again, is due to the post-network transfer copy overheads associated with *Offload-Callback* design. The sudden increase in overlap in the large message region for *Offload-Staged* can be attributed to variability of compute kernel used to measure overlap potential in the IMB benchmark suite.

### 4.3.5 Latency performance of MPI *Ialltoall* operation

Figure 4.18 shows the latency of an MPI *Ialltoall* followed directly by an MPI *Wait* on 16, 32, and 64 GPU nodes respectively. In other words, we evaluate the performance of NBC operation with the blocking counterpart and metric. In addition to good overlap, the proposed *Offload-GDR* and *Offload-Callback* deliver a good latency and outperform the *Offload-Staged* scheme by a factor of 3.5X and 2.7X respectively on 16 GPU nodes. On 64 nodes, the benefits of the proposed designs reduce due to the bandwidth saturation. Indeed, the Alltoall communication pattern is a dense communication pattern where the network bandwidth is limiting the performance at large scale.

### 4.3.6 Overlap/Latency performance of MPI *Iscatter* operation

The overlap potential of the different schemes using *Iscatter* operation on different system sizes are depicted by Figure 4.19. The comparison trend is mostly similar at different scales. The exception to this is that the message size at which *Offload-Callback* and
Offload-Staged start to show improved overlap potential decreases with scale. This is because the growth in the cost of completing network transfers outweighs the growth in the cost of the device to memory transfer at the root of the scatter. As expected, Offload-GDR shows a constant and good overlap potential up to 82%. Interestingly, Offload-Staged shows better overlap performance over Offload-Callback which is best explained in the latter’s case by the possibility of cudaMemcpyAsync-based transfer not completing by the time compute phase finishes and thus requiring some assistance from the host during the MPI_Wait phase. In contrast, in case of Offload-Staged the complete transfer from device to host precedes the network activity and hence provides better overlap at the cost of latency as shown in Figure 4.20.

### 4.3.7 Overlap/Latency performance of MPI_Igather operation

Figures 4.21 and 4.22 depict overlap and latency performance of medium/large message sizes for the Igather operation using the different offload-based designs on 16, 32 and 64 GPU nodes respectively. In contrast to the overlap trend, the Offload-GDR and Offload-Callback deliver better latency and needing 2,100 µs and 5,950 µs on 32 nodes with 1 MB message size respectively. This leads to an improvement of 9X and 3X compared to the Offload-Staged design respectively. Similar trends are seen with 16 and 64 GPU nodes configurations as well. The overlap concern with Offload-GDR and Offload-Callback on further examination showed that this was an artifact of direct algorithm implementation. The algorithm was implemented as a series of receive operations from a source and wait from source operations before receiving from the next. This results in excessive use of task_wait operations in the task list which affects overlap. An optimization that can be
proposed here is to allow for single wait task for multiple completion upon completion of all receives being posted.

### 4.3.8 Overlap/Latency performance of with more than 1 MPI process-per-node

Figures 4.23 and 6.3 show the overlap and latency when running two MPI processes per node with each using its own GPU and IB rail. The trends are similar to those for the single MPI process scenarios and this demonstrates the scalability of the proposed designs. The only outlier here is the case with MPI_Scatter where the benefits of using \textit{Offload-Callback} is shown over \textit{Offload-GDR}. Scatter requires reading all the data from root and the HCA faces bandwidth limitations in reading directly from the GPU which is circumvented when \textit{Offload-Callback} is used as shown in Figure 6.3(b).

### 4.3.9 On Latency comparison with blocking variants and on the effects of running GPU workloads

Figures 4.25(a) and 4.25(b) show that, as expected, running a GPU workload can have a noticeable effect on \textit{Offload-Callback} but nearly no effect on overlap and performance of \textit{Offload-GDR} when using MPI_Iallgather. Figures 4.26(a) and 4.26(b) show that performance of proposed designs come close to achieving the same latency achieved by blocking counterparts for the case of Alltoall but has higher latency in comparison with Allgather. This is best explained by the fact that Alltoall performance hinges on the network bandwidth but for Allgather the blocking variant which uses RDMA transfers has a lead over tasklists based on send-recev operations.
Figure 4.17: Overlap of MPI_Ialltoall
Figure 4.18: Latency of MPI_Ialltoall
The figure illustrates the overlap of MPI scatter operations for different node counts and message sizes. The graphs show the overlap percentage for three different node counts: 16, 32, and 64 nodes. Each graph includes three types of offload strategies: Offload-Callback, Offload-Staged, and Offload-GDR. The x-axis represents the message size, while the y-axis indicates the overlap percentage. The graphs demonstrate how the overlap varies with message size and the number of nodes, providing insights into the efficiency of different offload strategies.
Figure 4.20: Latency of MPI_Iscatter
Figure 4.21: Overlap of MPI_Igather
Figure 4.22: Latency of MPI_Igather
Figure 4.23: Overlap of MPI-Ialltoall, MPI-Iscatter, MPI-Igather with 64 nodes (2PPN)
Figure 4.24: Latency of MPI_Ialltoall, MPI_Isscatter, MPI_Igather with 64 nodes (2PPN)
Figure 4.25: Effect of Compute Location on Latency/Overlap

(a) 64 Nodes-1PPN: MPI_Iallgather (Overlap)    (b) 64 Nodes-1PPN: MPI_Iallgather (Latency)

Figure 4.26: Latency Comparison with blocking variant

(a) 64 Nodes-1PPN: MPI_Ialltoall    (b) 64 Nodes-1PPN: MPI_Iallgather
4.4 Related Work

The use of multicast features for improving collective communications, and the associated performance improvements have been well studied in the past. In [10], Mamidala et. al. proposed a hybrid design for MPI broadcast, allreduce, and barrier collective operations, coupling hardware multicast and shared memory communication. The authors also proposed a dynamic attach policy to alleviate the possible processing skews. In [38], the authors studied the applicability of using hardware multicast features for efficiently creating MPI communicators dynamically. Liu et. al proposed scalable designs for MPI broadcast with reliability [34]. Hoefler et. al [22, 47] have proposed designs for scalable broadcast and novel algorithms using hardware multicast, and achieved near constant time broadcast performance for large scale clusters. Wang et.al first proposed the idea of performing point-to-point operations from GPU buffers in MVAPICH2-GPU using pipelined mechanisms [74]. Potluri et.al proposed designs to directly perform MPI operations from GPU buffers using GDR [54]. With regard to collectives, the use of pipelined Alltoall and Allgather operations on GPU buffers was studied by Ashish et. al [62]. The rCUDA framework offers GPU virtualization through the use of custom calls replacing CUDA calls which can be used by machines across clusters as well [16]. Lean et al. in [45] proposed concepts that allow an HCA to access GPU memory similar to GDR. However, their concepts require specific hardware and cannot be applied to production ready HPC systems. Venkatesh et al. proposed to mix hardware multicast with GDR to design efficient broadcast operation for GPU based systems [69].

Overlapping computation communication with NBC has traditionally been a topic of great interest. Given the dominant nature of the MPI programming model in the scientific computation domain, it is likely that the key driver for the acceptance of this interface by
the application community will be the real benefits offered by intelligent MPI designs and implementations of NBC. Sancho et al. [59] studied the benefits of using dedicated processors for progressing the global reduction operation and study the benefits of overlapping the MPI_Iallreduce operations. Hoefler et al. proposed using host based techniques for designing non-blocking collective operations [8]. However, host based techniques offer limited performance portability and may not deliver complete overlap. Kandalla et al. proposed servlet-based design for MPI-NBC support. Subramoni et al. proposed communication primitives for blocking collective operations with the CORE-Direct [64]. While all these designs for host based communications exist, there is no known work to design efficient NBC for GPU-enabled systems.

4.5 Summary

In this chapter, we presented the design for an efficient GPU data broadcast operation (that is near scale-invariant for short message sizes) enabled by hardware multicast feature in InfiniBand in tandem with the GPUDirect RDMA feature on NVIDIA GPU-equipped InfiniBand clusters. We proposed a methodology to overcome the performance challenges posed by UD transport requirement of hardware multicast in designing broadcasts from GPU buffers using IB architecture’s scatter-gather-list abstraction. We designed synthetic benchmarks to mimic communication patterns of streaming applications and use them to measure the benefits of the proposed designs. The results of our experimental evaluation show up to a 60% decrease in latency of the operation compared to a Host-staged multicast scheme for a broadcast that involves 64 GPU nodes. In a synthetic streaming benchmark, we also show a 3X-4X decrease in execution of a 1,000 back-to-back broadcasts that reflects live data dissemination, again for a 64-node broadcast. This work can be extended to
include efficient MPI broadcast, allreduce and allgather operations that can take advantage of GPUDirect RDMA and hardware multicast.

Also in this chapter, we presented the first study and use of Non-blocking MPI collective operations (NBC) from GPU memories. We proposed truly asynchronous designs for GPU NBC through a combination of Core-Direct communication offload mechanisms from Mellanox and NVIDIA’s GPUDirect-RDMA capabilities. Finally, to circumvent the well known peer-to-peer limitations, we proposed a novel design that leverages CUDA’s callback functionality to offer good latency and overlap. Through the proposed designs, we showed close to 100% overlap for dense collective operations such MPI_Ialltoall and MPI_Iallgather that use GPU buffers in the medium and large message range. We also showed that such overlap can be achieved without much sacrifice in the latency of these non-blocking operations. The techniques discussed in this chapter are applicable to MPI_Ireduce and MPI_Iallreduce along with an exploration of hierarchical GPU NBC.
Chapter 5: Achieving CPU-GPU Control Plane Decoupling Using Device-initiated Network Operations

In this chapter, we first describe the overheads incurred in the critical path when CPU explicitly manages synchronization between inter-dependent MPI communication and GPU routines to ensure dependency satisfaction between compute and communication phases. Also, we highlight how such synchronization overheads can adversely affect performance, particularly in strong scaling application scenarios. With this, we discuss the envisioned model to achieve CPU-GPU decoupling and propose a set of minor semantic extensions that help leverage decoupling capabilities in MPI applications. Then, we describe the details of GPUDirect-aSync (GDS), a novel technology developed to initiate network operations from within the GPU that helps avoid the CPU from having to keep track of the progress of MPI communication and GPU routines. Finally, we propose a set of point-to-point protocols that honor both MPI semantics (with extensions) and CUDA stream semantics while simultaneously ensuring good performance and reducing the impact of the aforementioned synchronization overhead. Experimental evaluation with micro-benchmarks and an application-kernel mimicking benchmark indicate that the proposed designs reduce the majority MPI-CUDA phase synchronization and signifies the potential achievable with full applications.
5.1 Overheads From CPU-managed Synchronization Between CUDA Compute and MPI Phases

In MPI+CUDA applications, compute regions are offloaded onto the GPU using CUDA and inter-GPU data movement is expressed using MPI. However, synchronizing calls in CUDA (cudaStreamSynchronize, cudaDeviceSynchronize, etc.) and MPI (eg: MPI_Waitall) are used to satisfy potential dependencies between compute and communication phases in the application. These inter-phase synchronizations incur software/hardware overheads when launching CUDA and network operations as they need CPU intervention to manage the two activities. Ultimately, they translate to losses in performance and energy usage of the application.

In state-of-the-art applications, CUDA streams (independent execution sequences on the GPU) are used to concurrently schedule independent work onto the GPU in order to overlap computation with communication and to hide offload latencies. Figure 5.1 provides the pseudo-code for how optimized stencil codes are typically implemented. Communication-independent interior computation is separated out and launched on one stream (step 1 with an interior stream in the example). Boundary data is packed using a kernel launched on another stream (step 2 with boundary stream). The CPU waits for the pack to complete, using a cudaStreamSynchronize operation. The boundary data is exchanged between GPUs using MPI (step 4). Once the MPI data exchange is complete, unpack and boundary compute kernels (steps 5-7) are launched on the boundary stream. Steps 2-7 are expected to happen concurrently with step 1 providing computation-communication overlap and latency hiding. However, steps 2 to 7 are executed sequentially and the CPU has to be actively involved in each step through blocking synchronization calls to schedule the next phase. This pattern is representative of several MPI+CUDA applications such as QUDA [11], HPCMG [43].
for (timestep = 0; ...) {
    compute_interior_kernel <<<...,interior_stream>>> (…)
    pack_kernel <<<...,boundary_stream>>> (…)
    cudaStreamSynchronize(boundary_stream)
    MPI_Irecv(…)
    MPI_Isend(…)
    MPI_Waitall(…)
    unpack_kernel <<<..., boundary_stream>>> (…)
    compute_xboundary_kernel <<<..., boundary_stream>>> (…)
    compute_yboundary_kernel <<<..., boundary_stream>>> (…)
    cudaDeviceSynchronize(…)
}

Figure 5.1: Pseudocode for 2DStencil CUDA+MPI Test

Figure 5.2 shows profiles for the stencil code discussed above. Each profile has a timeline (left to right) and has three rows (top to bottom) which show the activity on the CPU, interior stream and boundary stream, respectively. The horizontal bars are marked with numbers corresponding to the steps in the pseudocode in Figure 5.1. Today, as the CPU explicitly makes calls to synchronize the compute and communication phases on GPU clusters, there are two key overheads in the current GPU computing infrastructures. We use the stencil code example to highlight these:

- **Wastage of CPU Cycles:** The profile in Figure 5.2(a) shows how the activities on the two streams are overlapped. The CPU spends all of its time on launching work onto the GPU, making MPI calls for communication and waiting for the GPU work to complete. The overheads can be separated into two parts: (1) latencies (for offloading CUDA work and for MPI communication) and (2) polling (waiting for GPU
work to complete). In Figure 5.2(a) where the interior compute kernel execution time is significant (commonly observed in weak scaling), the CPU wastes a larger chunk of its cycles polling for GPU to complete. Though both GPU and the network (IB in this case) provide mechanisms to offload computation and communication, respectively, the CPU still has to block in each step to coordinate between the GPU and the network. This results in wasting CPU cycles in polling for completions of either the GPU compute phase or the IB communication phase.

![Figure 5.2: Offload Overheads dominate with strong scaling as CPU-GPU synchronization becomes more frequent](image)

- **Wastage of GPU Cycles and CUDA Launch Overhead Limitations:** Figure 5.2(b) shows the profile when the domain size per GPU is 256x256. This is representative of the use case when applications are strong scaled on GPU clusters. Here, we see that most of the CPU time is attributed to latencies. A little overlap is achieved between activity on the interior stream and the boundary stream because of the short kernel execution times. As the computation on the GPU cannot be scheduled until
CPU completes the communication phase, the whole offload latency for the unpack kernel launch shows up in the critical path. Similarly, because of the synchronization at the end of each stage, the offload latency for interior compute kernel also shows up in the critical path. These latencies significantly limit the overall runtime of the application and the utilization of the GPU in such latency-limited strong scaling scenarios. In addition to CPU resource wastage shown earlier, we can clearly see, the wastage of GPU resources as we are not able to keep the GPU computing due to the synchronizations to be performed by the CPU.

Both the issues arise from the fact that CPU has to be involved when the application transitions between computation and communication phases, i.e. whenever GPU and the network have to be synchronized.

5.2 Mitigating Synchronization Costs Stream-based MPI: Extensions

5.2.1 General Vision For Control Flow Offload to GPU

Our overall vision is to allow the GPU to be able to initiate a communication operation (say, for an *MPI_Send*) after a CUDA Kernel is completed, or to be able to start a new kernel as soon as a communication operation completes (say, after an *MPI_Recv*). The goal, however, is to achieve this without the CPU having to explicitly monitor completion of either of the preceding operation. In other words, the goal is to alleviate the CPU from having to wait for completion of the CUDA kernel before issuing an MPI_Send that potentially reads buffers accessed by the kernel. Alternatively, the goal is to alleviate the CPU from having the wait for the MPI_Recv to complete before issuing a kernel launch that potentially accesses the received buffer. To achieve this, we take advantage of CUDA streams. A CUDA stream (independent command queue) can be used to describe list of
tasks to be performed by the GPU in the order they were issued. We envision enqueuing even MPI communication operations on to CUDA streams alongside kernel launches and other CUDA calls. In principle, doing this, allows the GPU to track the progress of both CUDA and MPI operations and thus guarantee dependency satisfaction between compute and communication phases by following stream order without taking up too many CPU cycles. For this, however, MPI calls need to be minimally extended and their semantics need to be defined for operability with CUDA Stream semantics. Here, we discuss these extensions and semantics that allow applications to make use of GDS technology with minimal code changes as envisioned in Figure 5.3.

Figure 5.3: Blocking Stream-based Send/Recv Semantics
5.2.2 Semantics of Issuing MPI Operations on Streams

Traditional MPI programming model and semantics include blocking and non-blocking operations. These semantics have been defined with regards to the CPU issuing the MPI calls. A blocking operation prevents CPU cycles from being used until the communication buffer is ready for use/reuse. Even for non-blocking operations, the MPI model associates a blocking operation (like MPI\_Wait) to ensure reusability of buffers. On the other hand, CUDA Stream-based operations are non-blocking from the CPU’s perspective by default. Hence, issuing MPI operations as part of a CUDA stream requires even traditional blocking MPI operations to be non-blocking from the CPU’s perspective. This is also desirable from the standpoint of saving CPU cycles. In other words, all MPI operations (with the stream extensions), including MPI\_Wait for instance, are non-blocking from CPU’s point of view. Thus the CPU returns as soon as it posts and offloads the operation to the stream. While the operations are non-blocking from CPU perspective, they follow the CUDA stream semantics and ordering. An MPI operation with stream extensions returns to CPU context as soon as it is posted, however, its actual execution will depend on the operations enqueued onto the CUDA stream preceding the MPI call and the original semantics of the MPI call itself. For instance, a send operation posted after a kernel launch on the same stream will return immediately from the CPU perspective. The actual data movement, however, will be triggered by the GPU as soon as the kernel completes. The next subsequent operation enqueued onto the same CUDA stream, say another GPU kernel, will be launched only after the send buffer is available for reuse. On the other hand, consider an alternate scenario. The issue sequence consists of 1. a GPU kernel (kernel A) on CUDA stream (stream A), 2. a non-blocking send on stream A, 3. GPU kernel (kernel B) on stream A. In this scenario, due to the non-blocking nature of the MPI operation in step 2, kernel B does not wait for
the completion of the MPI call. Thus, MPI operations with stream extension retain their traditional semantics from the CUDA stream’s perspective. We note that if an MPI operation is used without stream extensions, it keeps all its traditional semantics. Further, in the same application, a process can use both semantics interchangeably. Here, we restrict the scope to the discussion of point-to-point operations but we note that the discussions apply to collective and RMA routines.

### 5.2.3 Alternatives for Stream Association with MPI Operations:

An intuitive way of associating a stream object with an operation such as *MPI_Send* is to include an additional parameter to specify the stream object as shown in Listing 5.1. This approach allows an MPI operation the freedom to choose any of the asynchronous streams that the application is handling. The requirement, however, is that application changes have to be made for the different MPI calls with an extra parameter to be passed.

Listing 5.1: MPI_Send With Stream Extension

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, CUstream *stream);
```

Another alternative is to associate a stream object to a communicator and all operations issued using the communicator would offload the MPI operations onto the GPU stream instead. To add the stream to a communicator, we treat the stream as a field that we pass to the communicator during its creation or duplication (potentially through MPI Info). The advantage of this approach is that all MPI communication calls require no change as they inherently make use of the stream if the communicator is tagged with a CUDA stream and hence code changes will be minimal. The disadvantages include the number of communicators to manage for applications using a lot of communicators and further this restricts the scope of the number of streams that a particular communicator can use to 1. Regardless
of the association method, the new semantics of MPI operations on the stream remain the same.

5.2.4 Informal Definition of Semantics of MPI Operations on Streams:

Here, we go over specific semantics of important blocking and non-blocking operations on streams.

5.2.4.1 Semantics of Send and Receive Operations on Stream:

An MPI Send issued with an associated stream awaits for the completion of all previous CUDA operations issued on that stream and blocks all further stream operations until the buffer passed as an argument to the call is ready for reuse. An MPI Recv issued with a stream also awaits pending CUDA operations on the stream to complete and blocks all subsequent stream operations until a matching send (match composed of the communicator, rank, and tag) is found. As mentioned earlier, in addition to the data movement semantics of MPI, we honor the ordering and concurrency semantics of stream. Hence, the result of operations which issue transfers from the same buffer on two different concurrent streams are undefined. As all these operations are now non-blocking from the CPU perspective, next, we define progress semantics calls to provide asynchronous progression and notify the CPU about the completion of the different operations issued.

Non-blocking operations such as MPI_Isend and MPI_Irecv still wait for a completion of CUDA operations issued on the stream prior to their call but they do not stall the stream from resuming subsequent CUDA work until a corresponding MPI_Wait is called. All other semantics are similar to regular MPI counterpart. For instance, a process issuing a sequence such as Kernel_A, MPI_Isend, and Kernel_B on the same stream will observe
an overlap between the data movement of the Isend operation with the execution of the Kernel B.

5.2.4.2 Progressing Communication Calls Offloaded to the GPU:

In principle, while the proposed model and semantics allow for the MPI operation to be completely progressed on the GPU without CPU intervention, the matching requirement that MPI specifies requires some CPU assistance. This arises as current hardware does not provide matching support and all existing hardware follow a queuing-based system where work requests are consumed and fulfilled in the order that they are posted. To facilitate CPU assistance in orchestrating communication protocols used to realize the proposed stream-based communication operations, two alternative approaches are possible: 1) Interception of `cudaStreamSynchronize` calls by the MPI runtimes to ensure the completion of the different MPI operations on that stream. It is worth noting that our model aims to reduce the number of such calls to maximize the asynchronous progress. However, depending on the application requirement at least one call is needed (before finalize) to ensure the completion of the stream operations. 2) Similar to the interception method, we introduce an explicit operation `MPI_Wait_stream_completion` (that takes a single communicator argument) which ensures that all MPI operations issued on the communicator (which has a stream associated with it) prior to the call are completed before returning from the call. Internally, this operation will perform `cudaStreamSynchronize` before returning. Hence a sequence such as `Kernel_A, MPI_Send`, and `Kernel_B` will simply issue basic operations required to start the compute and communication sequence but progress and completion necessities required to realize this, is embedded in a subsequent `MPI_Wait_stream_completion` call. In essence, this avoids repeated kernel launch overheads that are incurred in the existing MPI+CUDA programming model.
Table 5.1: Semantics of Stream-based MPI Communication

<table>
<thead>
<tr>
<th>Operation</th>
<th>Launch after previous stream instruction completes</th>
<th>Block next stream instruction completes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>MPI_Wait(Send request)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>MPI_Wait(Recv request)</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

5.3 Mitigating Synchronization Costs Stream-based MPI: GPU-initiated Communication Designs using GPUDirect-aSync (GDS)

In order to offload communication and synchronization tasks to the GPU, a new system-level interaction between the GPU and HCA components is required. A communication operation can be split into two parts: 1) preparation of the operation, and 2) issuing/starting of the operation. The preparation of the operation is instruction intensive and furthermore, the GPU execution thread is very slow in executing sequential instructions. Hence, in order to avoid a slow critical path, we rely on the CPU to prepare the network operations which are later issued by the GPU. This necessitates a system-level interaction among the CPU, the GPU, and the HCA. Here, we describe a prototype implementation of GPUDirect-aSync and the different system-level enhancements required to enable such an interaction.

5.3.1 Overview of CPU, GPU and HCA interaction under GDS

The general idea is to use the CPU to prepare and queue communication work to the IB HCA but to delay triggering of the operation by not ringing the doorbell. The act of ringing the doorbell is converted into a list of primitives offered by the GPU Scheduling Unit (GSU) and queued onto a stream. This allows GPU to trigger the start of a communication operation in stream order when prior compute work on the stream completes. Similarly, the operation of polling for an IB completion is converted into GSU primitives and queued
onto a stream. This blocks any later activity submitted on the stream until polling completion is notified. Figure 5.5 shows the architecture of stream queues and scheduling unit on the GPU. To realize these two fundamental operations of triggering communication and polling for completion, enhancements at the CUDA-level, OS-level and IB-verbs-level are needed. The software stack (see Figure 5.4) is partitioned into a mid-level library, libgdsync. This library offers a set of IB Verbs-like stream-oriented interfaces such as gds_stream_queue_send and gds_stream_wait_cq. gds_stream_queue_send provides a way to have the send operation triggered upon the completion of preceding CUDA operations. gds_stream_wait_cq provides a way to have the stream blocked until a completion event is generated on the completion queue passed as an argument to the call. Those call down respectively to new IB extended verbs and the new memory oriented CUDA API extensions.

Figure 5.4: Extended CUDA/InfinBand software stack for GPUDirect aSync
5.3.2 CUDA API Enhancements

The GSU exposes new generic memory oriented primitives (exposed as CUDA API extensions) that allow enqueuing operations on to a CUDA stream that either affects the content of memory addressed in the calls or affects the stream based on the operation. These operations include:

- **WRITE32** to write a 32-bit word, optionally enforcing an ordering with respect to subsequent writes (barrier or flush).
- **WAIT32** to wait (polling) on a 32-bit word with a constant payload, using three different predicates (greater-or-equal, equal, and bitwise AND).
- **FLUSH** to enforce GPU memory consistency, as observed by GPU internal hardware processing units.

These APIs (Listing 5.2) produce new types of work, in addition to CUDA kernel launches and memory copies, which the GSU will fetch and process in stream order. For example, a stream could be made to write a particular memory location specified in the WRITE32 command with a value specified in the command when preceding instructions in the stream have completed. Likewise, a stream could be made to wait for a value (as specified in the argument to the command) to appear at a memory location as specified in the WAIT32 command. As the memory location specified in these commands passed to the stream could take as argument a memory location mapped from other PCIe devices like the HCA (through OS-level enhancements), they can be used to realize polling on CQ and “ringing the doorbell” on the GPU’s part.

Listing 5.2: CUDA API Enhancements

```c
cuStreamWaitValue32(CUstream stream, CUDeviceptr addr,
```

113
5.3.3 OS-level Enhancements

Extensions are needed in kernel-mode in order: 1) to support the placement of IB Verbs completion queues (CQ) on GPU memory (for subsequent polling) and 2) to enable GPU
peer mappings, that is the ability to access the PCIe resources (i.e. the BARs) of a device through the direct memory mappings on the GPU MMU (for triggering send operations, for example). The latter enables passing addresses corresponding to the HCA (e.g. the doorbell) registers to the CUDA memory APIs, which would otherwise generate an access violation error. The above two kernel-mode extensions, in turn, allow for the realization of `gds_stream_wait_cq` and `gds_stream_queue_send` respectively.

### 5.3.4 IB Verbs Enhancements

A new verbs primitive, `ibv_exp_post_send_ex`, which fills the HCA work queue with the WQEs and returns a descriptor list with the information necessary to trigger those operations is used by `libgdsync`. `gds_stream_queue_send` converts this list into a sequence of WRITE32 and FLUSH operations which are enqueued on to the CUDA stream (passed to it as an argument) using `cuStreamBatchMemOp`. This fulfills the triggering action needed to issue send operations in stream order.

Waiting on CQE is the most challenging part for the GPU. In GDS, the concept of a CUDA stream waiting on a CQE is actually split into two parts: 1) the GSU polls on a particular DWORD belonging to the CQE using WAIT32, detecting updates coming from the HCA; 2) later the CPU cleans up/poisons that CQE in `ibv_poll_cq`. This fulfills the polling action needed to block the stream until send or receive operations complete.

There is a potential race if these two steps are not executed in order, as the GPU may be stuck in WAIT32 if the CQE is cleaned up by the CPU before the GPU has a chance of observing it being signaled. To avoid that, the MPI notification handling / CQ processing framework must maintain a dedicated tracking logic, effectively inserting an additional
step between 1 and 2, where the GSU signals (WRITE32) that the particular CQE has been processed.

5.4 MPI-level Designs for Decoupling CPU-GPU Control Flow

Now, we describe how calls exposed by *libgdsync* are used together with CUDA API (including extensions) and InfiniBand functionality to realize MPI operations on streams whose semantics were informally defined in Section 5.2.

5.4.1 Protocols for Enqueuing MPI Point-to-point Communication Operations onto Streams

All MPI operations offloaded onto a stream are non-blocking from the CPU perspective. The MPI operations, however, do honor the semantics of CUDA streams and can block the GPU execution. Issuing operations on the stream acquire a notion of enqueuing work referred to as *enqueue phase calls*, and progressing/monitoring stream operations and its completion at a later point is referred to as *progress phase calls*. We exploit functionality exposed by CUDA runtime and GDS for the design of the traditional eager and rendezvous MPI protocols as indicated by the acronyms as shown in Table 5.2.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Acronym</th>
<th>Operation</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>CudaMemcpyAsync</td>
<td>CuMemAsync</td>
<td>send_complete_event</td>
<td>SndCmpEvnt</td>
</tr>
<tr>
<td>stream_queue_send</td>
<td>StQSend</td>
<td>recv_complete_event</td>
<td>RcvCmpEvnt</td>
</tr>
<tr>
<td>stream_wait_cq</td>
<td>StWtCQ</td>
<td>recv_ready_event</td>
<td>RcvRdyEvnt</td>
</tr>
<tr>
<td>cudaEventRecord</td>
<td>CuEvntRcrd</td>
<td>cudaStreamWaitEvent</td>
<td>CuStWtEvnt</td>
</tr>
<tr>
<td>application_stream</td>
<td>AppSt</td>
<td>cudaStreamWriteVal</td>
<td>CuStWrVal</td>
</tr>
<tr>
<td>helper_stream</td>
<td>HlpSt</td>
<td>cudaStreamWaitVal</td>
<td>CuStWtVal</td>
</tr>
</tbody>
</table>
5.4.2 Eager Protocol

To realize the eager transfer of messages, most MPI runtimes pre-allocate and register buffers with the network card for direct memory access. Depending on the protocol, these buffers may be located either on host memory or GPU memory. When beneficial to do so, user data is either copied to these buffers or received data is transferred from these buffers to user memory in an eager manner. Now, we discuss alternative designs for both sender and receiver sides and use Figure 5.6 to describe the primary eager methodology.

![Eager Design Diagram](image)

**Figure 5.6: Eager Design**

5.4.2.1 Eager Sender Side Alternatives:

Depending on whether the protocol uses the eager buffers (EBUF) at the sender side or not, different designs can be adopted.
**EBUF protocol:** With the assumption that the eager buffers (EBUF) are being used, *enqueue phase calls* can include an asynchronous CUDA-assisted copy (*CuMemAsync*) on the application stream from user buffer to eager buffer. Any operation issued on the stream will not start until *CuMemAsync* completes according to CUDA semantics. This is followed by a *StQSnd* operation on the eager buffer and a *CuEvntRcrd(AppSt, SndCmpEvnt)* operation. These two operations help to ensure that the stream has arrived at a point where send buffer is ready to be sent and to check if the buffer is ready for reuse respectively. If *CuMemAsync* is from user buffer to an eager buffer on the host, the send operation specifies a single address location. If the eager buffer resides on the GPU device, the send operation must specify two address locations: 1) for the packet details generated on the host memory (as generating them on the GPU is cost-prohibitive), and 2) for the user buffer on GPU. The subsequent send operation can use *IB gather-abstraction* to realize that this is a single send operation.

**UBUF protocol:** Alternatively, an implementation can choose not to use the eager buffers at the sender and directly send from the User buffer (UBUF). In this case, *enqueue phase calls* can omit *CuMemAsync* and directly register the user buffer with the HCA before issuing *StQSnd* operation and *StWtCQ* followed by a *CuEvntRcrd(AppSt, SndCmpEvnt)*. The use of *StWtCQ* ensures that subsequent operations enqueued in the stream do not touch send buffer until send has completed (as signaled by the generation of the completion event associated with it). *Progress phase calls* can include querying if *SndCmpEvnt* has completed and polling the associated completion queue to remove the completion entry generated by the HCA.
5.4.2.2 Eager Receiver Side Alternatives:

On the receiver side, if at the time of preparing/enqueuing `MPI_Recv`, it is noted that the matching sent message is already available then a simple `CuMemAsync` call is issued as part of `MPI_Recv`. However, if the matching is not found, then the situation is more complex as enqueue phase calls cannot include instructions that result in receiving sent data to the user specified receive buffer until the stream has arrived at a point where all previously issued operations on the CUDA stream have completed (defined as a safe point). Hence, we propose a LoopBack-based design (LB). In LB, receiver-side enqueue phase calls includes `CuEvntRcrd(AppSt, RcvRdyEvnt)` call followed by `ibv_post_recv` from self and `StWtCQ` on receive completion queue associated with source rank. This way the specific receive operation can be satisfied when progress conditions are met as discussed later. Please note that, during initialization of a stream-based communicator, loopback connections, regular all-to-all connections, and separate completion queues are established in order to circumvent the case of unblocking a CUDA stream incorrectly — which can occur if common completion queue is used. Alternatively, instead of the `StWtCQ` operation, `CuStWtVal` can be issued which serves the same purpose which allows for variations during progress phase of the receive operation.

Receive-side progress phase first involves awaiting for the matching send data from an `MPI_Send` to arrive. Note that the matching operation is performed by the CPU. When matching has occurred, `RcvRdyEvnt` is queried before issuing a loopback send operation or a `CuMemAsync` and `CuStWrVal` to destination user receive buffer to satisfy either the `StWtCQ` or `CuStWtVal`, respectively. This realizes receive completion and frees the stream to move on to subsequent enqueued operations. It should be noted that instead of the
send-receive operation pairing, rdma-write-with-immediate can be used instead (immediate being required to generate a completion event on which the receiver is waiting in the StWiCQ call).

5.4.2.3 Non-blocking Send/Recv Eager designs:

Non-blocking send and receive operations should not block the application stream from progressing unless a corresponding Wait operation on the stream is issued.

Naively following the same design as the blocking operations, i.e. offloading the MPI operations solely to the application stream, may lead to a deadlock. Consider the scenario where two processes $P_0$ and $P_1$ issue the sequence of $MPI_Irecv$, $MPI_Isend$ and $MPI_Waitall$ on $stream_0$ and $stream_1$ respectively in the same order at both ends. While this is semantically correct, the non-overtaking nature of operations issued on a stream is susceptible to deadlocks. Using the following protocol leads to a deadlock — issuing an $ibv_post_recv$ on the application stream (i.e $stream_0$ for $P_0$ and $stream_1$ for $P_1$) followed by $StWtCQ\,(recv\_request,\,app\_stream)$ for $MPI_Irecv$; issuing $StQSnd$ followed by $StWtCQ\, (send\_request,\,app\_stream)$ for $MPI_Isend$. This is because neither of the processes can reach the point where $StQSnd$ can be issued as it waits for the previous $StWiCQ$ to complete. Further use of operations such as $StWiCQ$ and $StQSnd$ on the application stream may need to be avoided as they redundantly block operations as in the sequence of $MPI_Irecv$, $kernel\_A$ where $kernel\_A$ is allowed to begin without the completion of $MPI_Irecv$.

To avoid such potential deadlock, we propose a Helper Stream(HlpSt)-based protocol. A non-blocking send can use $StQSnd$ operation. For the corresponding Wait, $StWtCQ$ can be used. On the receiver side, if a matching send is found, a $CuMemAsync$ can be issued. If this is not the case, then during the enqueue phase, a $StWiCQ$ operation on a helper stream is posted along with $CuMemAsync$ over the helper stream from a buffer
location which contains user sent data to receiver specified receive buffer. This is followed by \textit{CuEvntRcrd(HlpSt,RcvCmpEvnt)}. During \textit{Wait}, \textit{enqueue phase} involves issuing \textit{CuStWtEvnt(AppSt,RcvCmpEvnt)}. Receiver-side \textit{progress phase} is similar to blocking approach where a send operation is issued from sender sent buffer location in order to satisfy pending \textit{StWtCQ} and \textit{CuMemAsync} operations. A helper stream used by the library avoids the potential deadlock owing to the separation of operations. We provide more details on the usage of the HlpSt-based protocol when discussing the design for Rendezvous protocols. It should be noted that the use of a helper stream does have a consequence of sequentializing all operations on the communicator even though the operations may potentially be executed concurrently. This can be avoided through the use of a pool of helper streams.

5.4.3 Rendezvous Protocol

Rendezvous protocol is used to avoid using scarce eager buffer resource on the receiver side. As in the case of traditional rendezvous protocol, control messages are exchanged between sender and receiver to realize the direct transfer of data from sender to receiver using GDR or pipeline designs [53]. As mentioned previously, a receive operation posted on receive buffer must be satisfied only when the receiver stream has arrived at a safe point. Hence the proposed rendezvous protocol makes use of \textit{RDMA-Read} operation to read sender’s data directly into receiver’s buffer. On the sender side, \textit{enqueue phase} involves issuing a \textit{StQSnd} of an RTS (request-to-send) message and \textit{StWtCQ} for a FIN (finish) over the application stream. Sender-side \textit{progress phase} involves simple book-keeping operations. On the receiver side, a helper stream is made use of to avoid potential deadlock scenarios with non-blocking variants as previously discussed. Receiver-side \textit{enqueue phase} involves issuing \textit{StQSnd (RDMA-Read), StWtCQ (send_cq for source rank)} and \textit{StQSnd}
(FIN) operations over the application stream if matching send is found at the time of enqueuing. If not, CuEvntRcrd(HlpSt, RcvRdyEvnt), CuStWtEvnt(HlpSt, RcvRdyEvnt), and CuStWtEvnt(AppSt, RcvCmpEvnt) are called in enqueue phase. In progress phase, StQSnd (HlpSt, RDMA-Read) and StWtCQ (HlpSt, send_cq) are called to awaiting receiver stream arrival before fetching sender’s data. Further to release the sender, StQSnd (HlpSt, FIN), StWtCQ (helper_stream) followed by CuEvntRcrd(HlpSt, RcvCmpEvnt) are called to release the application receiver stream. Figures 5.7 and 5.8 highlight the different steps in realizing the proposed GDS-aware MPI rendezvous protocol.

Figure 5.7: Sender-Side Rendezvous Design
5.5 Performance Evaluation

5.5.1 Experimental Testbed and System Software

Here, we describe our experimental setup and evaluate the design presented in Section 5.4. Owing to the preliminary phase of the framework being used, the testbed in the work is very minimal and includes two compute nodes with Intel Sandy Bridge series of processors using Xeon Dual quad-core processor nodes operating at 2.6 GHz with 32 GB RAM. Each node is equipped with MT26428 FDR ConnectX HCAs (56 Gbps data rate) with PCI-Ex Gen2 interfaces. In addition, each of the nodes is composed of a single Tesla K20c GPU. The goal of this evaluation is to provide a proof of concept of the proposed new technology and its associated runtime designs rather than a thorough scalability evaluation. Experiments on this setup are performed using one MPI process per node using a single
GPU each. That said, however, we expect the benefits of such design to be significant at scale.

### 5.5.2 Micro-benchmark Results

We demonstrate the results of two classes of experiments for this work.

**Latency Benchmark:** This benchmark consists of a sender and a receiver MPI process on two separate nodes which transfer data located on GPU memory. We perform two variations of evaluations with the latency experiments: 1) The sender and receiver issue `MPI_Send` and `MPI_Recv` on an asynchronous CUDA stream respectively. This is followed by a `MPI_Wait_stream_completion` call on both processes. This experiment helps us to understand the raw cost of issuing stream-based send/recv operations. 2) Prior to issuing the `MPI_Send`, a GPU kernel is launched on the sender side, while on the receiver side the `MPI_Recv` is followed up by a kernel call. The kernel used on both ends are the same and use the same CUDA stream as the MPI operations. The kernel call itself and execution consume a fixed duration of time on both ends and this experiment helps us to understand the impact of stream-based calls kernel execution time and vice versa. A simple increment operation kernel is used for the experiments with the size of the array to increment is equal to the message size used in either the send or receive operation used for that specific test. The tests show, however, that the kernel invocation call time is not affected by the size of the argument for range tested (1-byte to 512 Kilobytes).

**Throughput Benchmark:** The throughput experiments are composed of a fixed count (256) of `MPI_Send` and `MPI_Recv` on an asynchronous CUDA operation on the sender and receiver side, respectively. This is followed by an `MPI_Wait_stream_completion` call at both ends. The variant of this experiment is similar to the latency benchmark where a kernel is
issued prior to each $MPI_{Send}$ on the sender side and a kernel is issued after each $MPI_{Recv}$ reusing the same stream as the MPI operations. These experiments help us to evaluate the possible benefits when kernel and MPI operations can be batched together and completed en masse in the application. This is expected to be the main use case for our proposed design framework.

While there exist possibilities of enabling different design components discussed in Section 5.4 for both sender and receiver sides, we primarily show results with when design alternatives have noticeable performance differences. For eager-protocol evaluation, we compare copy-based and registration-based designs at high-level but we further show the impact of using $cudaStreamWriteVal/cudaStreamWaitVal$ as indicated by $(writeval)$ in Figures 5.11 and 5.12. For rendezvous transfer evaluation, we consider just the Read-based protocol. Furthermore, we show the mean value of results for the experiments in this work as most experiments (repeated in the order of 1000 X 10 iterations) showed very little variation. One exception to this case being the throughput experiment involving the registration-based eager transfer which shows up to 4% variation on the receiver side as shown in Figure 5.10(b) which is likely due to the contention of PCIe resources for inter-node and intra-node loopback data transfers.

**Latency of Stream-based MPI Point-to-point Operations:** Figure 5.11 depicts the performance of the different design schemes for the latency benchmark. Comparing Figures 5.11 (a) and 5.11(b), it can also be seen that the performance of registration-based eager transfers have shorter latency (about 14 microseconds for less than 512-byte range) in comparison with the copy-based eager transfers (about 25 microseconds for less than 512-byte range). The impact of enqueuing a kernel operation (either before/after) with an MPI call can also be seen in these two figures where the latency of launching a kernel
(about 12 to 15 microseconds) has a noticeable footprint on the overall time of the measured sequence of operations. This, however, is expected given that the micro-benchmark calls repeated iterations of \((\text{Kernel} + \text{MPI\_Send} + \text{MPI\_Stream\_wait\_completion})\) or \((\text{MPI\_Recv} + \text{Kernel} + \text{MPI\_Stream\_wait\_completion})\) and hence there is no room to amortize the initial kernel invocation cost. There are some minor improvements observed from not using \text{cudaStreamWriteVal/cudaStreamWaitVal} indicated by \text{writeval} in the range of messages
Figure 5.11: Send/Receive Latency
Figure 5.12: Send/Receive Throughput
between 128 bytes and 2 Kilobytes for both the eager design variants. For the large message range (Figure 5.11(c)), the transfer cost dominates and hence the launch overhead is imperceptible.

**Throughput of Stream-based MPI Point-to-point Operations:** It can be seen that for the short and medium message range message (Figures 5.12 (a) and 5.12 (b)), there is an upper and lower band of lines representing the performance of the sender and the receiver side latency costs. These differences stem from the use of loopback operations on the receiver end which are expected to incur higher costs owing to the larger number of network operations needed to realize the receive operation in the general case (i.e., when matching-send is not found in the unexpected queue). This is much more noticeable than the latency experiments due to the throughput stress that the benchmark places on the network adapter in this experiment. From this experiment, it is also clear that some time overhead due to kernel launches can be reclaimed with the use of GDS in throughput scenarios. This is observed from the fact that the difference between an MPI operation and (kernel+MPI) operation is about 8 to 9 microseconds which are an improvement of about 30% at the sender side and close to complete reclamation at the receiver side (Figures 5.12 (a) and (b)). At the receiver-side, there is a significant impact in using `cudaStreamWriteVal/cudaStreamWaitVal` which indicates the potential drawback in using memory-mapped calls. For bandwidth-bound rendezvous transfers, there is nearly no impact in making kernel invocations alongside (Figure 5.12(c)).

**Application Pattern Simulation and Comparison with MPI+CUDA** The throughput benchmark we have worked on indeed mimics the application usage of stream-enabled MPI operations in MPI+CUDA applications. We use the copy-based eager transfer protocol for this experiment as it shows better average (sender + receiver) latency and throughput.
characteristics in comparison with the registration-based scheme. In Figure 5.13, we show how the copy-based eager transfer scheme shows over 30% improvement in comparison with a similar benchmark composed of regular MPI_Send and MPI_Recv operations with CUDA kernel and stream synchronize calls before the MPI_Send and after the MPI_Recv, respectively. These benefits are mainly due to minimizing the polling overhead, hiding the kernel launch overhead, and keeping the GPU busy for the entire application run. This highlights the effectiveness of GPU handling communication operations that satisfy stream dependencies.

![Comparison of GDS-based Send/Receive Operations with GPU kernels with Traditional MPI+CUDA Application Kernel](image)

Figure 5.13: Comparison of GDS-based Send/Receive Operations with GPU kernels with Traditional MPI+CUDA Application Kernel
5.6 Related Work

Several works have been published in literature that makes use of novel RDMA technologies and use of network offloading methods in communication with MPI stacks. Matt et. al proposed methods to express small message reduction operations on Quadrics offload units [28]. On the other hand, Hoefler et.al proposed MPI extensions to express non-blocking collective operations from an MPI context required for a higher level abstraction network-offloaded MPI collectives. Several subsequent works have been proposed to make use of offload technologies on InfiniBand networks [8].

Other works have looked expressing MPI operations on buffers that do not reside in host memory. Hao et. al [74] transparently achieve this by using Unified Virtual Architecture (UVA) extensions, which is introduced in CUDA 4.0. On the other hand, methods to leverage novel RDMA technologies in communicating GPU memories, they forgo the use of host memory as a temporary buffer space before issuing requests. Other works have looked at the necessary extensions of MPI datatype interoperability with such CUDA-based MPI calls [61]. Potluri et. al proposed for the first time the use of GPU-Direct RDMA (GDR) with MPI operations that allow the network controller to directly touch GPU memory in order carry out inter-node communication [53]. Related works have also looked at offloading the communication and initiating the communication from GPUs [14, 46]. Unlike GDS which offloads only the initiation to the GPU, [14, 46] offload both parts of an operation (preparation and starting) to the GPU which severely limits their performance due to the slow GPU sequential execution.
5.7 Summary

NVIDIA GPUs are changing the landscape of HPC systems. CUDA-Aware MPI runtimes are pushing the frontiers of performance and productivity on such systems. While CUDA IPC and GPUDirect RDMA features provide a direct data movement path between GPUs and HCA bypassing the CPU, the control flow, and the synchronization between different computation, communication, and synchronization phases still requires the CPU intervention. This intervention requirement places the CPU in the critical path and significantly limits the asynchronous behavior of the GPU as it introduces overheads. In this chapter, we tackled such limitations and proposed a new technology called GPUDirect-aSync (GDS) to decouple the control flow between the CPU and GPUs. It provides a way to offload the control flow to GPU and removes the CPU from the critical path of satisfying CUDA compute and CUDA-MPI communication dependencies. Using GDS, we extend the behavior of MPI point-point communication calls. Alternative designs for the different MPI point-point eager and rendezvous protocols with the new GDS behavior are provided and analyzed. The evaluation of our proposed MPI designs shows a 30% improvement using an application-kernel pattern simulating benchmarks when using the communication operations in the short and medium message range thus showing good promise for inclusion in MPI+CUDA application usage.
Chapter 6: Designing Application-Oblivious and Energy-Efficient MPI Two-sided Communication Runtime

In this chapter, based on the drawbacks highlighted about existing state-of-the-art methods of saving energy in MPI programs (refer Chapter 1.2), we present methods to design an energy efficient runtime that uses intricate knowledge of point-to-point and collective protocols to automatically save energy during MPI routines without application hints. Furthermore, we describe methods of deciding whether application of energy levers results in true positives and false positives. Through experimental evaluation, we show that the proposed methods can save energy for a plethora of applications/application-kernels with negligible performance degradation.

6.1 Energy-Aware-MPI (EAM) Solution Space

6.1.1 Terminology and Assumptions

Table 7.1 shows the symbols which are used for communication modeling of the EAM runtime. Let \( \mathcal{P} \) represent the set of MPI processes, where \( p_i \in \mathcal{P}, 0 \leq i < |\mathcal{P}| \). Let \( \mathcal{L} \) represent a set of power/energy levers such that each \( \mathcal{L}_i \in \mathcal{L} \) is a triple \( \{\delta_i, \gamma_i, \psi_i\} \mid 0 \leq i < |\mathcal{L}| \), where \( \delta_i \) is the time threshold after at which power lever \( i \) is used; \( \gamma_i \) is the overhead of using the lever and \( \psi_i \) is the power improvement. The levers are sorted in non-decreasing value of \( \delta \).
We define slack ($s$) to be the actual time spent by an MPI process in a single MPI call such as send, receive, wait or collective communication operation. A repeated execution of the same MPI primitive can potentially generate a different value of slack owing to change in application critical path or owing to ambient network and system conditions. We use LogGP model [9] to predict the minimum expected communication time ($w$) of an MPI primitive. LogGP model makes the following assumptions: 1) tightly connected sender/receiver and 2) silent/dedicated network. When either of these conditions are invalid, slack deviates from the minimum expected communication time. Since expected communication time as per LogGP is a lower bound on slack (due to constants based on hardware limits), in each case: $w \leq s$.

Each power lever incurs a time overhead, $\gamma_i$, when used. Examples of power levers are DVFS and core-idling. Hence, a power lever should be used only when the $\gamma_i$ can be amortized over slack. We use a user-acceptable overhead ($\rho$) to calculate the time threshold

<table>
<thead>
<tr>
<th></th>
<th>Definition</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Process Set</td>
<td>$\mathcal{P}$</td>
</tr>
<tr>
<td>2</td>
<td>Power/Energy Lever Set</td>
<td>$\mathcal{L}$</td>
</tr>
<tr>
<td>3</td>
<td>Network Latency</td>
<td>$l$</td>
</tr>
<tr>
<td>4</td>
<td>Message Size</td>
<td>$m$</td>
</tr>
<tr>
<td>5</td>
<td>Network Bandwidth</td>
<td>$\frac{1}{\tau}$</td>
</tr>
<tr>
<td>6</td>
<td>Overhead term in LogGP</td>
<td>$o$</td>
</tr>
<tr>
<td>7</td>
<td>Copy overhead for $m$</td>
<td>$c_m$</td>
</tr>
<tr>
<td>8</td>
<td>Registration overhead for $m$</td>
<td>$r_m$</td>
</tr>
<tr>
<td>9</td>
<td>Slack</td>
<td>$s$</td>
</tr>
<tr>
<td>10</td>
<td>Expected Communication Time</td>
<td>$w$</td>
</tr>
<tr>
<td>11</td>
<td>Overhead of $\mathcal{L}_i$</td>
<td>$\gamma_i$</td>
</tr>
<tr>
<td>12</td>
<td>Power improvement of $\mathcal{L}_i$</td>
<td>$\psi_i$</td>
</tr>
<tr>
<td>13</td>
<td>Time Threshold for using $\mathcal{L}_i$</td>
<td>$\delta_i$</td>
</tr>
<tr>
<td>14</td>
<td>Degradation permitted</td>
<td>$\rho$</td>
</tr>
</tbody>
</table>

Table 6.1: Symbols used for communication and energy modeling of proposed MPI runtime
(δᵢ) after which a lever may be used. Essentially, \( \delta_i = \frac{\gamma_i}{\rho} \). As an example, for core-idling with \( \gamma = 5\mu s \), if \( \rho = 0.05 \) then \( \delta = 100\mu s \). When \( s \geq \delta_i \), then each \( \mathcal{L}_j \) such that \( \delta_j \leq \delta_i \) can be used for saving power.

In EAM, the expected communication time is calculated analytically and slack is measured online. A lever is applied after its threshold (\( \delta_i \)) has been crossed (Figure 6.1(a)). For several MPI primitives, expected communication time itself may be larger than (\( \delta_i \)) of a power lever, in which case the lever(s) are applied at the start of the MPI call (Figure 6.1(b)).

![Figure 6.1: An example of using two power levers in EAM](image)

### 6.1.2 Optimistic and Pessimistic MPI Runtimes

A performance-only optimized runtime, which is the de facto in most MPI libraries, represents an optimistic execution. Essentially \( \forall \mathcal{L}_i \in \mathcal{L}, \delta_i \leftarrow \infty \). The primary assumption in optimistic execution is that MPI calls do not present enough opportunity for applying a power/energy lever. When slack is large, optimistic execution can waste energy. Alternatively, an energy-only optimization implies that power levers can be applied at the start.
of each MPI call. This approach is referred as pessimistic execution. Specifically, for pessimistic execution, $\forall L_i \in L, \delta_i \leftarrow 0$. This approach has the potential to reduce energy consumption, especially when $s \geq \delta_{|L|-1}$, in which case all power levers can be applied. In many other cases, when slack is small, pessimistic execution can severely degrade performance and increase energy consumption.

The objective of proposed Energy Aware MPI (EAM) runtime is to find an optimal point, where appropriate power levers can be applied without degrading performance. EAM achieves this objective by using a combination of in-depth analytical modeling of communication primitives, and online slack observed. It uses the combination to apply power levers during appropriate points in an MPI call. The next section is concerned with taking common point-to-point communication primitives and their protocols to achieve this objective.

### 6.1.3 Point-to-Point Primitives

Point-to-point communication primitives (send and receive) can be classified as blocking and non-blocking calls. A return from a blocking point-to-point communication primitive implies that the buffer may be re-used. MPI semantics do not provide any other guarantees for blocking call. Non-blocking communication primitives only provide a request object in return, and the buffer cannot be re-used unless progress made on the request handle indicates otherwise.

An in-depth look into MPI primitives and protocols is necessary to calculate the expected communication time. MPI runtime typically use eager protocol for small messages and rendezvous protocols for large messages. The section below describes the calculation of expected communication time for send and receive with the eager and rendezvous
communication protocols. Using the estimated communication time, each section also
discusses the selection of appropriate power levers and their effects on performance and
energy savings.

6.1.4 Send using Eager Protocol

Using LogGP model, the expected communication time \( w \) for small messages is \( (l +
o + m \cdot G) \approx l \), since \( m \) is small. However, MPI completion semantics only guarantee that
a buffer may be re-used at the return of a blocking MPI send call. Several MPI runtimes,
including EAM, indicate early send completion by simply copying the user-buffer to an
intermediate communication buffer, resulting in an expected time \( w = c_m + o \) (\( c_m \) is the
time to perform an intra-process memory copy). Typically, \( c_m + o \ll l \). In practice, \( \gamma > l \).
As an example, \( \gamma \) for core-idling (also referred to as interrupt-driven execution for the rest
of the chapter) is \( 5 \mu s \), while latency is \( \approx 1 \mu s \) for several modern interconnects. As a result,
in the general case, the Eager-Send becomes a likely false positive for applying a power
lever (Figure 6.3(a)).

However, early send completion is not always possible, specifically when a spare inter-
mediate communication buffer is not available. In this case, EAM uses alternative proto-
cols. In one case, EAM registers the user-buffer and initiates the communication resulting
in an expected communication time of \( r_m + l \approx r_m \), since \( r_m \gg l \). In practice, \( r_m \gg \gamma \) for
a power lever such as interrupt-driven execution. This case becomes a true positive when
\( \exists L_i \in L \mid \delta_i \leq r_m \).

6.1.5 Receive using Eager Protocol

As presented in Section 6.1.4 above, point-to-point communication using eager proto-
col does not synchronize the sender and receiver. A send may arrive before a matching
receive is posted. In this case, EAM enqueues this unexpected message in an *unexpected queue*. When matching send is absent in the unexpected queue, a blocking receive waits for a matching send to arrive. The expected time in receive-processing is dependent upon whether the peer send is in the unexpected queue.

To calculate expected communication time in EAM, the unexpected queue is searched for a possible matching send. If a matching send is present in the unexpected queue, the payload can be directly copied to user-buffer \( w = c_m \). Assuming a small queue search time, this is a false positive case, since the copy cost of an eager message is small.

However, when a matching send is not present in the unexpected queue, it is non-trivial to calculate the expected arrival time. MPI semantics allow an application to post larger receive buffer than matching send. A communication model, which uses the size of posted receive would overestimate the communication time — a potential for false positives. EAM addresses this limitation by using empirically observed slack in this case, and applying the power levers as their corresponding threshold \( \delta \) is crossed. A detailed state transition is presented in Figure 6.3(b). Note that the reception of an eager-send message at the receiver is effectively a FIN message (Section 6.1.6). This approach minimizes perturbation of application’s execution time, albeit at loss of some energy savings.

### 6.1.6 Send using Rendezvous Protocol

Rendezvous communication protocol is used for large messages. It implicitly synchronizes the sender and receiver. To accelerate large message communication, modern interconnects use Remote Direct Memory Access (RDMA), which allows a process to read/write memory of another process without its involvement. RDMA may be initiated
from sender-to-receiver (write) or receiver-to-sender (read) resulting in RDMA-Write and RDMA-Read based rendezvous protocols, respectively.

**RDMA-Write Based Rendezvous Protocol** RDMA-Write based rendezvous protocol (shown in Figure 6.2(a)) uses a combination of control messages and a payload message to complete the data transfer. The expected communication time is:

\[
(l + o + m_{RTS} \cdot G) + (l + o + m_{CTS} \cdot G) + (l + o + m \cdot G) +
\]

\[
l + o + m_{FIN} \cdot G. \quad \text{Since control messages are small, } w = (4 \cdot l + m \cdot G) \quad (o \ll l). \quad \text{However, this time is a lower bound for several reasons: The expected time derived above assumes that the sender and receiver are well synchronized, which may not be the case depending upon the application characteristics. In addition, EAM uses a lazy de-registration technique which stores buffer information and its registration data structure in a small cache. When cache miss occurs, the buffer needs to be registered on-the-fly, resulting in expected communication time } w = 4 \cdot l + m \cdot G + \sum_{\text{sender}} r_m + \sum_{\text{receiver}} r_m. \quad \text{As presented earlier, this becomes a likely case for applying power lever.}
\]

Once sender and receiver have arrived in the protocol, the expected communication time is \(\Omega(2 \cdot l + m \cdot G)\) for both sender and receiver. Hence, all \(L_i \in \mathcal{L} \mid \delta_i \leq 2 \cdot l + m \cdot G\) can be used as the appropriate lever for energy efficiency. Since control messages (RTS, CTS and FIN) are small, the sender uses the Send-Eager design for sending RTS and FIN messages and Receive-Eager design for receiving CTS message. Figure 6.3(a) shows the state transitions for a sender in RDMA-Write based protocol.

**RDMA-Read Based Rendezvous Protocol** Figure 6.2(b) shows the processing steps in RDMA-Read Based Rendezvous Protocol. When sender and receiver are well synchronized, expected communication time is \(3 \cdot l + m \cdot G\) (Without loss of generality, \(G\) for RDMA-Read and RDMA-Write is considered equal). In EAM, the sender uses Send-Eager
6.1.7 **Receive using Rendezvous Protocol**

**RDMA-Write Based Rendezvous Protocol** A receiver in this protocol receives RTS and FIN control messages (Fig 6.2(a)). For receiving the RTS message, EAM uses the *Receive-Eager* design. Similarly, *Send-Eager* design is used for sending the CTS message.

Once the RTS is received, expected communication time for the FIN message can be calculated 

\[ w = \frac{l}{\text{CTS}} + l + m \cdot G + \frac{l}{\text{FIN}}. \]

When the sender uses blocking communication,
\(w \approx s\). However, slack can deviate significantly, if the sender uses non-blocking communication, in which case it would be able to send the payload and FIN message, when it makes the MPI progress call. The state transitions are presented in Figure 6.3(b).

**RDMA-Read Based Rendezvous Protocol** In this protocol, the receiver expects an RTS message, which is handled by Receive-Eager design in EAM. The receiver sends the FIN message, which is handled using the Send-Eager design. Once RTS is received, the receiver can make independent progress on reading the payload from sender’s memory by using RDMA. The expected communication time can be calculated as:

\[
w = l + m \cdot G + \frac{\text{payload}}{\text{FIN}}.\]

Hence \(L_i \in L \mid \delta_i \geq 2 \cdot l + m \cdot G\), \(L_i\) can be used as the power lever for saving energy. The state transition diagram for the receiver following RDMA-Read based rendezvous protocol is shown in Figure 6.3(d).

### 6.1.8 Non-Blocking Messages

MPI applications use non-blocking messages to facilitate overlap of communication with computation by just initiating the request but not necessarily completing it. Non-blocking calls return a request handle to the user. EAM optimizes non-blocking messages by performing minimal protocol processing by deferring the protocol processing to progress primitives. For eager protocol, EAM uses aggressive techniques such as simply copying \((w = c_m)\) or enqueuing the message in a queue. Non-blocking messages which use rendezvous protocol in EAM, only initiate a control message. The receiver (upon calling non-blocking recv) searches the unexpected queue and only initiates either the CTS (for RDMA-Write) or the data read (for RDMA-Read). In all cases, the expectation is that the time spent in protocol processing does not warrant using a power lever. Hence, non-blocking messages are considered a false positive for power efficiency.
6.1.9 Progress Primitives

MPI provides several progress primitives, which can be classified in blocking (such as Wait, Waitall, Probe, etc) and non-blocking (such as Iprobe, Test, Testall, etc) categories. Non-blocking progress primitives typically look at the internal data structures corresponding to request handle, and return appropriate status. All these are false positives for using a power lever.

Blocking progress primitives require that at least one request handle must complete before the control returns to the caller (For Waitall, all request handles must complete). EAM maintains a queue of request handles (composed of non-blocking send and receive calls) which have not completed yet. To conserve power during progress primitives, EAM uses a timer based approach. At the start of an MPI progress primitive, EAM starts a timer and makes progress on the request handle(s). If a request completes, the timer is reset. Otherwise, as the time spent without a successful completion in the primitive increases, additional power levers are used as their corresponding thresholds are crossed.

This simple yet effective approach used by EAM is able to minimize the false positives since power levers are used carefully. In many cases, when processes are not well synchronized, EAM is automatically able to use the slack for conserving energy.

6.1.10 Group Communication Primitives

These primitives can be classified as blocking and non-blocking collectives. Here, we primarily focus on blocking collectives. Blocking MPI collective communication primitives can be further classified as rooted (broadcast, reduce, gather, scatter, etc) and un-rooted (allgather, alltoall, allreduce, etc) primitives. In EAM, we consider each of these categories for power optimizations.
(a) RDMA-Write: Sender’s Eager and Rendezvous transition rules

(b) RDMA-Write: Receiver’s Eager and Rendezvous transition rules

(c) RDMA-Read: Sender’s Eager and Rendezvous transition rules

(d) RDMA-Read: Receiver’s Eager and Rendezvous transition rules

Figure 6.3: EAM State Transitions
6.1.11 Rooted Collective Communication Primitives

Broadcast and reduce are the primary rooted collective operations used in MPI applications. They are typically implemented using k-nomial tree communication structure. Rooted primitives do not synchronize all the processes. This property of these primitives allows a process to complete the operation after it has received a message from its parent (except for root), and has forwarded to its children (an inverse property is observed for reduce, where messages start at the leaves and complete at the root). Due to this property, a natural skew is observed in rooted collectives. For broadcast, each process \(p_i\) receives the message at

\[
T(i) = \begin{cases} 
  t_{pt2pt}(\lceil \log_k(|P|/i) \rceil) & \text{if } i \in k^p, \ p \in \mathbb{Z} > 0 \\
  t_{pt2pt}(\lceil \log_k(i) \rceil) - 1 + T(|k^{\lceil \log_k(i) \rceil}) & \text{otherwise}
\end{cases}
\]

where process \(p_i\) has MPI rank as \(i\), and \(t_{pt2pt}\) is the time for point-to-point communication. In EAM, this communication model is used to see if this model can be used to select a power lever at the start of the collective operation. For tree-based operations, as the root uses non-blocking sends, the non-roots know the lower-bound time after which they will receive the data. The actual slack can be much higher at scale, due to different arrival times of various processes in the collective operation. Hence, as the slack increasingly deviates from expected communication time, EAM uses applicable power levers for energy savings. Similarly for reduce, an inverse communication pattern is used, which is used in EAM for saving power.

6.1.12 Un-rooted Collectives

All-to-all Broadcast All-to-all Broadcast uses several algorithms such as recursive doubling for small messages, and ring algorithm for large messages. Here, we look at the ring algorithm. Let \(p_i \in \mathcal{P}\) be the process to arrive last in the operation, then \(s_{p_{i-1}} \gg\)
\( s_{p_j}, \forall p_j \mid j = 0 \cdots |\mathcal{P}| - 1, j \neq i. \) (Notice that if \( p_i \) arrives late, its communication partner \( p_{i-1} \) observes the maximum slack).

As a property of the ring, each process can complete the first step except \( p_{i-1} \) and \( p_i \). Hence, \( p_{i-1} \) observes slack in the first step, which can be used for energy efficiency. However, during the second step, communication in the ring is stalled till \( p_i \) arrives in the operation. As a result, each process can use the slack for energy efficiency. Once \( p_i \) arrives in the collective, \( w \approx s \). This information is used in EAM to maximize true positives for energy efficiency. When \( t_{pt2pt} \) is large enough for using a power lever, point-to-point communication models presented earlier automatically provide energy efficiency.

**All-to-all Personalized Exchange** All-to-all Personalized exchange is used in matrix transpose and kernels such as Fourier Transform. The expected communication time of the *de facto* pairwise-exchange algorithm is \( t_{pt2pt} \cdot |\mathcal{P}| \). In this primitive, the true positives are captured by using point-to-point communication modeling at each step. At each step, EAM uses the property of the algorithm to determine whether the participating process has arrived in the collective or not. This information is gathered using an indirect method which uses the current step and rank of the participating process to determine the skew of processes, which have already communicated with the participating process. This information is used in the following steps to decide whether a participating process has arrived in the collective, and determine whether \( s \approx w \).

### 6.1.12.1 Other Collective Communication Primitives

MPI supports a large number of collective communication primitives and several algorithms for each primitive. By default, these algorithms are designed using point-to-point communication primitives. Hence, power/energy optimization for point-to-point primitives are able to capture the energy efficiency as much as possible.
<table>
<thead>
<tr>
<th>Application</th>
<th>Major MPI calls</th>
<th>Class</th>
<th>Benefits(%)</th>
<th>Loss(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>miniFE</td>
<td>Allreduce</td>
<td>Iterative/Temporal</td>
<td>25%</td>
<td>1%</td>
</tr>
<tr>
<td>miniMD</td>
<td>Create_cart, Barrier, Allreduce</td>
<td>Iterative/Temporal</td>
<td>26%</td>
<td>2%</td>
</tr>
<tr>
<td>miniGhost</td>
<td>Waitany, Allreduce</td>
<td>Iterative/Temporal</td>
<td>23%</td>
<td>0%</td>
</tr>
<tr>
<td>CloverLeaf</td>
<td>Allreduce</td>
<td>Iterative/Temporal</td>
<td>12%</td>
<td>3%</td>
</tr>
<tr>
<td>CoMD</td>
<td>SendRecv, Barrier</td>
<td>Iterative/Temporal*</td>
<td>4%</td>
<td>4%</td>
</tr>
<tr>
<td>Hoomd-Blue</td>
<td>Allreduce, Bcast</td>
<td>Iterative/Temporal</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>AMG</td>
<td>Allreduce, Allgather</td>
<td>Iterative/Non-Temporal</td>
<td>10%</td>
<td>1.15%</td>
</tr>
<tr>
<td>Sweep3D</td>
<td>Recv</td>
<td>Iterative/Non-Temporal</td>
<td>12%</td>
<td>1%</td>
</tr>
<tr>
<td>LULESH</td>
<td>Allreduce</td>
<td>Iterative/Non-Temporal</td>
<td>18%</td>
<td>0.5%</td>
</tr>
<tr>
<td>Graph500</td>
<td>Alltoall</td>
<td>Non-Iterative</td>
<td>41%</td>
<td>4%</td>
</tr>
</tbody>
</table>

Table 6.2: Summary of energy efficiency results with EAM runtime. Performance degradation and energy improvements are calculated using optimistic execution as baseline.

### 6.1.13 Bounds on Loss

#### 6.1.13.1 Bounds on Performance Loss

Maximum performance loss occurs in EAM when a message completion occurs immediately after a power lever is applied. If the communication completes at $\delta_i$, the overhead is $\gamma_i \delta_i$, which is $\rho$. Since $\rho$ is a user-acceptable overhead, even in worst case, the performance degradation is never beyond $\rho$.

#### 6.1.13.2 Bounds on Loss of Energy Savings

In EAM, the maximum loss of energy savings occurs, when a message completion precedes the use of a power lever. If the communication completes at $\delta_i$, the overall value of missed energy savings is $\psi_i \cdot (\delta_i - \delta_{i-1})$. However, in practice, the pathological case is rarely observed for reasonable values of $\delta$. 

146
6.2 Performance Evaluation

6.2.1 Highlights of Observed Results

Since we have 10 applications which are evaluated using a combination of scaling modes and processor counts, we provide a table to summarize the energy improvements and performance degradation of EAM relative to optimistic execution with classifications (Table 6.2).

Figures 6.4(a), 6.4(b), 6.5(a) and 6.5(b) show the relative energy and relative speedup of EAM and pessimistic execution to optimistic execution. A few general conclusions can be drawn from these results: 1. EAM yields energy savings of up to 41% and results in a performance loss of less than 5% for up to 4,096 processes. 2. As shown in Table 6.2, EAM handles a wide range of application classes whose performance is determined by a variety of MPI operations. 3. Use of EAM leads to the near-maximum energy savings achievable for the levers used during MPI communication (as MPIP profiles indicate) 4. The degradation incurred using EAM does not exceed the user set limitation of \( \rho = 5\% \) for every evaluated application on each process configuration.

For 5 applications - HooMD, miniMD, Sweep3D, AMG and Graph500 and one or more configurations, pessimistic execution provides energy savings, albeit with performance degradation while EAM avoids this. These figures validate the need for EAM, where it is still possible to achieve energy efficiency close enough to pessimistic execution in many cases, but without performance degradation.

However, it should be noted that MiniMD exhibits two cases where performance of EAM exceeds Optimistic execution. The primary reason is that using interrupt-driven execution uses loop-back instead of shared-memory copy, under which it is possible to overlap communication with computation. However, with shared memory based protocol,
CPU is involved in memory copy, which results in escalated execution time. The end result of this artifact is that there are energy gains that stem from both the application of the proposed energy rules as well as from the reduction in execution time. Hence the energy usage reduces beyond the expected 66% optimality (maximum savings for the case of interrupt-driven execution) communication expenditure.

6.2.2 Iterative/Temporal Applications

CloverLeaf: In a weak-scaling mode, the execution time for 512, 1,024 and 2,048 processes is 1672s, 1748s and 1756s, respectively. An average of 18% of total execution time is spent in MPI operations for 1,024 and 2,048 processes. The communication consists of MPI_Isend operations (13KBytes, which use eager protocol). While non-blocking sends are likely a false-positive, (as explained in design section), a significant slack is observed at receivers. The mpiP profile for 2,048 processes indicates that the average time spent by an MPI process is 18% and the communication time can vary from 7% (minimum)- 30% (maximum). Cloverleaf is an iterative/temporal application. However, it manifests itself as iterative/non-temporal, which is due to a combination of systems and work-imbalance across processes. In Cloverleaf, Allreduce is the dominant primitive, which takes 11% of the execution time, and overall energy savings are \( \approx 12\% \). This result indicates that EAM can achieve nearly perfect energy efficiency with interrupt-driven execution lever.

CoMD: We use a strong scaling mode with a grid size of 64x64x64. With strong scaling, MPI communication time increases and takes up to 20% (11% in Sendrecv), 30% (15% in Sendrecv) and 60% (36% in Sendrecv) of the total execution time with 512, 1,024 and 2,048 processes, respectively. The runs take 1.75s, 1.3s, and 1.2s, respectively. Communication consists of message sizes that decrease from 17 KBytes to 12 KBytes as process count
Figure 6.4: Speedup and Energy Consumption of Pessimistic and EAM relative to Optimistic (Default) approach for 512 and 1024 processes
Figure 6.5: Speedup and Energy Consumption of Pessimistic and EAM relative to Optimistic (Default) approach for 2048 and 4096 processes
increases. For each process count, the (maximum, average, minimum) MPI time varies as 
(22%, 15%, 20%), (33%, 25%, 30%) and (63%, 56%, 61%) for the 512, 1,024 and 2,048 processes, respectively. This indicates good load balancing and hence low values of slack which limits the opportunities for energy savings. EAM shows a maximum of 6% savings with 1,024 processes as shown in Figure 6.4(b).

![Figure 6.6: miniFE Allreduce duration](image)

*miniFE*: We used a problem size of 1024x1024x1024 with `-load_imbalance 20` option. This allows us to study the skew effect in the application using 20% load imbalance, while
testing the efficacy of EAM. MPI takes up 16% (of 56s), 15.6% (of 28s) and 18% (of 14s) of the total execution time for 512, 1,024 and 2,048 processes, respectively. Figure 6.2.2 shows the variation of Allreduce time spent by processes for different job sizes. As evident from Figure 6.6(a), a larger fraction of processes spend more time in MPI than other fraction, primarily due to load imbalance. This provides a greater energy saving opportunity in comparison to Figure 6.6(b) and 6.6(c) where this fraction is considerably smaller. For this reason, EAM provides diminishing energy savings of 24%, 7% and 6% for the different jobs with nearly zero performance degradation (Figures 6.4(a), 6.4(b) and 6.5(a)).

MiniGhost: Experiments are run with global grid size of size of 1024X1024X1024 with MPI time as 11% (of 31s), 16% (of 17s), 29% (of 10s) and 33% (of 5.3s) of execution time of 512, 1,024, 2,048 and 4,096 processes, respectively. The application is dominated by large message Isends (5.5MB - 1.4MB) with 8-byte Allreduces (19% for 4k-job) and WaitAlls (11% for 4k-job) consuming the most time. Despite good load balance (maximum MPI-time = 36.28%, minimum = 31.64%, average = 33.63%), the exclusive exchange of large messages provides an opportunity for 5%, 7%, 11%, 21% savings with EAM for different job sizes as shown in Figures 6.4(a), 6.4(b), 6.5(a) and 6.5(b) respectively.

6.2.3 Iterative/Non-Temporal Applications

Sweep3D: It is a neutron-transport application, which exhibits a wavefront communication pattern. Most MPI send and receive use eager protocol. As a result, pessimistic execution shows a major performance degradation by using power/energy lever on each MPI call. EAM eliminates the false positives, and triggers the use of power/energy lever only when the delay in receiving the message on the wavefront exceeds lever threshold.
The overall energy savings using 4,096 processes are 12% with a negligible performance loss - a clear case which demonstrates the need for EAM.

**LULESH:** This solves the sedov equation by using a 3-dimensional block decomposition of the data. The solution begins by initiating a force on the origin, and the impact on the material is studied using time-steps. Processes which are further away from the origin, receive the impact of blast much later in comparison to other processes. This hypothesis is validated by the fact that during synchronization (using Allreduce) processes spend 4% (minimum), 50% (maximum) and 6% (average) time — a significant variance. Allreduce takes \( \approx 27\% \) of execution time and EAM can achieve 20% energy efficiency (recall interrupt-driven execution provides 66% power savings). Pessimistic performs similarly since the time spent is Allreduce is significant, the overhead of pessimistic execution is not observed on relative speedup.

**AMG:** This application uses repeated coarsening and smoothening steps (V-cycle) during the iterative procedure. While iterative, the steps result in non-temporal communication pattern across processes. AMG is executed in strong scaling mode (total grid size = 256M elements). MPI takes up a maximum 45% (most in Waitall and Allreduce) for the default 7-point stencil with good scaling for 512 (13s), 1,024 (7s) and 2,048 (3.5s) processes. Due to the V-cycle, there significant load-imbalance is observed at scale, which allows for savings of up 25% (at 2% degradation), even when there are mostly short message (less than 4KB) transfers.

### 6.2.4 Non-Iterative Applications

**Graph500:** This benchmark uses an MPI implementation of the parallel Breadth-First-Search algorithm. The property of the algorithm results in a dearth of available parallelism
across processes. A majority of processes spend time waiting for work with dominant primitives as Alltoall and Alltoallv. Roughly 75% of the time is spent in MPI since less than 10% of the processes have useful work. The large value slack in the collective operation implies that EAM and Pessimistic are both capable for providing energy efficiency with insignificant performance loss, as indicated in the charts. EAM reduces the energy consumption by 50% (extracting maximum power savings possible with interrupt-driven execution) with 4% performance loss.

6.3 Related Work

Several methods to conserve energy in scientific applications have been proposed in the literature. In addition to [58], [23] and [66], Kappiah et al. use micro-operations-per-memory-load (UPM) to assign suitable frequencies for energy conservation [18]. This technique requires the use of code instrumentation, which can be difficult to repeat for large codes. Lim et al. proposed application-transparent methods of identifying regions and assigning appropriate frequencies within MPI programs to conserve energy [31]. The work requires empirical calculation of ‘closeness’ and ‘long enough’ parameters in addition to formulating a function that maps micro-ops-retired to suitable p-state(s). For many applications, this empirical calculation is difficult. Kappiah et al. target MPI processes that are not in the critical path for frequency scaling in order to arrive at an MPI call ‘just in time’ and hence conserve energy in programs that suffer from load imbalance [58]. Rong et al. proposed methods of utilizing temporal patterns at the coarse granularity of application while looking at the workload characteristics using counters to conserve energy using CPUMISER [19]. This work also proposed the use of API to demarcate regions of code with power throttling instructions. Green Queue — a framework for implementing
application-aware DVFS — which also leverages on workload imbalance among tasks to reduce the power consumption of MPI ranks with lighter workloads as well as reducing power consumption of specific phases within a single MPI process [50]. Hoefler et al. have recently proposed implementing energy efficient collectives by examining the different memory, runtime, and energy tradeoffs [21]. Kandalla et. al [23] and Sundriyal et. al [66] have both proposed ad hoc methods of conserving energy in specific regions within Alltoall collective by identifying idle phases. Similarly, Vishnu et al. have proposed automatically saving energy for bulk one-sided communication subsystems [72, 73], however, the benefits are demonstrated only for bulk communication transfer.

6.4 Summary

In this chapter, we have proposed an MPI runtime — Energy Aware MPI (EAM) — which provides energy efficiency without perturbing the application’s execution time. EAM is motivated by two notable trends in extreme scale systems: 1) Large number of workloads with vastly distinct communication characteristics as seen in iterative-but-temporal, iterative-but-non-temporal, and non-iterative patterns; and 2) System effects such as OS noise, and congestion/contention, due to which an application behavior becomes unpredictable.

EAM addresses these challenges by using a combination of communication models and empirical observations: 1) It uses application-oblivious communication modeling of MPI point-to-point (blocking/non-blocking), their protocols (eager/rendezvous), progress primitives and collective communication primitives to predict the expected communication time. When it is long enough, appropriate power lever(s) are used at the start of an
MPI call, maximizing possible energy efficiency. 2) When communication model under-predicts the actual time (slack) in the MPI calls (dilated due to algorithmic/system issues) additional power levers are automatically applied to save energy as slack increases. EAM is implemented using MVAPICH2, a high performance MPI on InfiniBand. We have evaluated EAM using ten MPI applications, dominated by different MPI primitives (such as Recv, Allreduce, Waitany, Alltoall). The performance evaluation using up to 4,096 processes shows 5–41% improvement in energy efficiency with minimal (less than 5% in all cases) performance degradation.
Chapter 7: Designing Application-Oblivious and Energy-Efficient
One-sided Communication Runtime

The use of one-sided communication calls such as Put, Get, and atomic operations such Fetch-and-add and compare-and-swap in distributed applications is steadily increasing owing to their ability to decouple data transfer from data synchronization and associated performance benefits. MPI 3.0 standard introduced a more complete set of one-sided operations with semantics that allow a great degree of flexibility in implementing these calls. However, due to this flexibility combined with the non-synchronous nature of applications using them, a large duration of MPI calls may be spent in waiting for communication progress. This provides scope to save a large amount of energy, especially in synchronization phases owing to the dynamic nature of applications (often irregular in nature) which use MPI remote memory access (RMA) calls. However, to realize automatic energy savings in applications without code changes, intricate details of the protocols used to implement one-sided communication and synchronization calls need to be understood. Furthermore, the different ways in which a range of one-sided calls interact with one another, especially in application scenarios, also needs to be understood. In this chapter, we first show the challenges in reasoning about the energy usage of RMA calls owing to semantic flexibility and associated implementation methods. Then, we generate such rules for energy savings within the the execution time span of one-sided communication and synchronization calls.
in a manner that allows co-existence with energy saving rules for 2-sided calls. Finally, we show the benefits of applying the proposed energy rules and show savings with one-sided applications and kernels.

### 7.1 Complexity of Analyzing Energy Usage of RMA Calls

The uncertainty in being able to predict the estimated time to complete an RMA call stems from the following sources:

- Owing to semantic flexibility and the freedom that RMA implementations exercise which can sometimes result in the actual transfer of data and deferred transfers on other occasions
- Due to hardware limitations, that programmers are often unaware of, that result in limitations for reduction operations, and when handling non-contiguous datatypes
- Unlike 2-sided calls, there are differences in the nature of RMA calls made by different ranks even when they are between the same synchronization calls
- Due to the dynamic nature of irregular applications that are most likely to use RMA calls
- Progress of RMA calls between the time of issue and the time of synchronization is unknown
- Due to system noise such as OS jitter, dynamic network traffic conditions

We discuss each of these sources in greater detail here. The decoupling of window creation, data movement and synchronization operations allows for a large degree of freedom in implementing MPI-RMA operations as allowed by MPI-3.1 standard. Figure 7.1 shows
a small sample set in realizing the common lock+accumulate+unlock combination of operations that can be used to realize shared counters, for instance. From a user’s perspective, this flexibility complicates the task of predicting both the time and energy footprint of each of the components or the whole set for that matter. In addition, the possibility of mixing and matching different window allocation semantics with the different data movement and synchronization primitives makes situation more complex. For instance, consider a runtime that implements window creation and all the data movement operations as non-blocking operations (as MPI semantics allow this). For such an implementation, these operations pose no potential for energy savings and applying any lever will result in a false positive, i.e falsely predicting that the application of a lower energy gear will result in positive energy savings. On the other hand, at the synchronization point, the completion of the data movement is blocked until the target process completes the window allocation. Hence even-though the window allocation may not be expected to have any impact on energy, its interaction with the data movement and synchronization may affect the energy footprint of the operations. Furthermore, depending on the synchronization method and the current state of processes involved in one-sided communication, some RMA data transfer calls result in immediate transfer of data while other times the transfer is deferred to a synchronization point. As this is highly dependent on the dynamic state of processes involved it is difficult to associate a specific static rule to a particular RMA call in the application without incurring false predictions.

Next, while most networks that support one-sided operations such as remote memory read and write operations over fairly large message sizes, this support for atomic memory operations such as fetch-and-op, compare-and-swap are generally limited to one or a few words. Despite this, calls such as MPI_Accumulate can executed with arbitrary message
sizes and this results in MPI implementations needing to resort to 2-sided realizations. If, however, the performance behavior and energy footprint of an accumulate for a single word or few words were to be extrapolated then straightforward predictions for accumulate operations with larger message counts would result in mispredictions.

2-sided calls such as MPI_Send and MPI_Recv, collective calls have a much greater degree of predictable participation. For instance, an MPI_Recv of a certain message size from a certain source often results in receiving a message of that size as a code are written with an MPI_Send of matching message size, even though the MPI semantics allow for flexibility at the receiver end. Collectives, by definition, have definite expectations about the size of message expected from each of the participating ranks involved in the collective. These expectations make it easier to predict the associated execution time for 2-sided calls. For 1-sided calls, however, the calls between the start and end of an epoch at different sites could be completely arbitrary and often is, owing to the irregular nature of applications that use RMA calls. Furthermore, the type of calls executed at different sites could be quite different and often results in different execution time of the synchronization phase at different sites.

To add to this, the progress of outstanding RMA calls when overlapped with compute routines cannot be ascertained owing to the nature of RMA protocols that follow semantics. For instance, in passive synchronization scenarios, passive side may be expected to participate in granting locks which may not allow any transfers to occur while computations proceed in the foreground.

Lastly, the net load on the system at different instances change and this has a non-deterministic delay added to most communication regardless of whether they are RMA calls or not. A high load on specific set of links which results in link contention, high load
on specific switches, high load on the NIC which is connected to the source of the RMA call can all lead to dilation of the data transfer time. Hence, predicting the time and energy expenditure of specific call tends to have a degree of misprediction from their hardwared defined lower bounds. Therefore, an in-depth analysis of the different RMA protocols and deriving rules to automatically save energy without degrading performance of RMA calls is necessary.

<table>
<thead>
<tr>
<th>Lock</th>
<th>Accumulate</th>
<th>Unlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Lock Request, Wait until</td>
<td>Issue hardware supported one-sided atomic op</td>
<td>Poll for completion, Issue</td>
</tr>
<tr>
<td>Lock Granted</td>
<td></td>
<td>Release Lock Request</td>
</tr>
<tr>
<td>Issue Lock Request</td>
<td>Wait until Lock Granted, Issue</td>
<td>Poll for completion, Issue</td>
</tr>
<tr>
<td></td>
<td>hardware supported one-sided atomic op</td>
<td>Release Lock Request</td>
</tr>
<tr>
<td>Issue Lock Request</td>
<td>Enqueue operation</td>
<td>Wait until Lock Granted, Issue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hardware supported one-sided atomic op,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Poll for completion, Issue</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Issue Release Lock Request</td>
</tr>
<tr>
<td>No-op</td>
<td>Enqueue operation</td>
<td>Issue Lock Request, Wait until</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lock Granted, Issue hardware supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>one-sided atomic op, Poll for completion,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Issue Release Lock Request</td>
</tr>
<tr>
<td>No-op</td>
<td>Enqueue operation</td>
<td>Pack lock-accumulate-unlock into one packet</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and send in</td>
</tr>
</tbody>
</table>

Figure 7.1: Non-Exhaustive Design Space Possibilities

7.2 Performance Impact from Using Pessimistic Energy-Aware Run-times

RMA usage model generally relies on relatively expensive synchronization calls followed by one-sided data transfer calls whose synchronization costs are already incurred.
Generally, this usage model leads to performance benefits gained from the one time synchronization cost that is amortized over the remainder of 1-sided calls. For such scenarios, it is not incorrect to assume that the use of a pessimistic energy-aware runtime will result in good energy savings, as they try save energy on every attempt to progress communication. If nearly synchronization call (where pessimistic runtimes) progresses multiple 1-sided communication operations, then owing to the relatively large amount of time spent in synchronization phase (where progress occurs) there is a high likelihood of saving energy with minimal performance degradation. This, however, is not the only usage model for RMA routines. There are application use case scenarios where synchronization and data transfer tend to be combined and often the combined phase does not consume a sufficient amount of time to potentially save energy after incurring the overheads of lower energy gear application. These scenarios can occur repeatedly in applications that rely on passive synchronization techniques. Use of a pessimistic energy-aware runtime in such scenarios can result in performance degradation with minimal or no energy savings. An example of this is in the implementation of hash tables using 1-sided calls where the sequence of $MPI\_Lock$, $MPI\_Get$, and $MPI\_Unlock$ may be repeatedly executed with very little time being spent in $MPI\_Unlock$ owing to shared lock usage. This further motivates the need for an energy-aware runtime that dynamically applies energy saving rules even in 1-sided scenarios.

### 7.3 Rules for Achieving Energy Efficiency in MPI-3 RMA Operations

In this Chapter, we visit some general protocols used to realize MPI-3 RMA window creation, transfer and synchronization operations. We identify potential hotspots where energy expenditure takes place without substantial contribution to overall program progress,
i.e where slack is abundant. We then describe general rules to apply an energy gear from a lever set $\mathcal{L}$ based on whether a shift in energy gear leads to a true positive, i.e when predicting that a shift to a lower energy gear results in actual positive energy savings. Later in Section 7.4, we cover concrete instances of deriving and applying energy rules for specific RMA protocols used in MVAPICH2 MPI stack. Similar to our previous work, we formalize rules using definitions from Table 7.1 [71]. Table 7.1 shows the symbols which are used for communication modeling of EAM runtime. Let $\mathcal{P}$ represent the set of MPI processes, where $p_i \in \mathcal{P}, 0 \leq i < |\mathcal{P}|$. Let $\mathcal{L}$ represent a set of power/energy levers such that each $\mathcal{L}_i \in \mathcal{L}$ is a triple $\{\delta_i, \gamma_i, \psi_i\} | 0 \leq i < |\mathcal{L}|$, where $\delta_i$ is the time threshold after at which power lever $i$ is used; $\gamma_i$ is the overhead of using the lever and $\psi_i$ is the power improvement. The levers are sorted in non-decreasing value of $\delta$. We define slack ($s$) to be the actual time spent by an MPI process in a single MPI call such as MPI_Put, MPI_Get, MPI_Accumulate or collective call such as MPI_Fence. A repeated execution of the same MPI primitive can potentially generate a different value of slack. We use LogGP model [9] to predict the minimum expected communication time or the lower bound ($w$) of an MPI primitive. Each power lever incurs a time overhead, $\gamma_i$, when used. Examples of power levers are DVFS and core-idling. Hence, a power lever should be used only when the $\gamma_i$ can be amortized over slack. We use a user-acceptable overhead ($\rho_i$) to calculate the time threshold ($\delta_i$) at which a lever may be used. Essentially, $\delta_i = \frac{\gamma_i}{\rho_i}$. As an example, for core-idling with $\gamma = 5\mu s$, if $\rho = 0.05$ then $\delta = 100\mu s$. When $s \geq \delta_i$, then each $\mathcal{L}_j | j \leq i$ can be used for saving power. With this premise, we try to generate rules on whether shifting the energy gear during the execution of an MPI call results in positive energy savings within acceptable performance degradation (true positive) or results in exceeding user permitted performance degradation with or without energy savings (false positive).
Table 7.1: Symbols used for communication and energy modeling of proposed MPI runtime

### 7.3.1 Energy Rules for Window Creation/Freeing

Window creation may involve a combination of allocating memory, registering it with the network device and exchanging access information (keys) with the processes group associated with the window abstraction (with the exception of `MPI_Win_create_dynamic`). The associated memory costs are generally small ($a_m, r_m$) and in the common case this changing energy levers for this call is a false positive for energy savings. However, due the collective nature (`allgather` needed for key exchange) of window creation, either a delay in entering the call on the part of a single process or a difference in size of window exposed at any single process may result in many processes having to wait for durations longer than $w = a_m + r_m + l$ (assuming $l + Mg \approx l$ for first transfer in `allgather` irrespective of allgather algorithm). Both these sources manifest as skews from the perspective of the allgather operation that precedes the completion of window creation and slack can be reclamied using previously established 2-sided MPI rules [71]. If the window creation is non-blocking
in nature and defers key exchange to a later data movement or synchronization primitive then the window creation call is treated as a false positive for energy savings.

Semantically, MPI info objects may seemingly nullify the need for such a rule if it provides the hint that all windows are similar sized (Info: same size) but load imbalance and system noise can present itself as skew regardless of the program specification and prolong collective call duration. Win.create.dynamic and Win.attach calls require that user explicitly ensure memory availability prior to access. Hence they are typically non-blocking in nature and a false positive for energy savings.

The rules for freeing the window (which may involve de-registering memory with the network device and freeing memory) is analogous as it must be realized using a collective (because it involves a barrier synchronization in order to ensure that lock/unlock operations are not called on a freed window) and hence previous collective rules may be reused [71].

### 7.3.2 Energy Rules for One-sided Calls and Synchronization

MPI-3 RMA transfer calls (such as Put, Accumulate, etc.) are non-blocking in nature and solely initiate the movement (if at all). Different synchronization calls (such as Fence, Unlock, Flush, etc) execute network operations in order to verify that the transfer has completed locally/remotely. Unlike the case with point-to-point and collective operations, the MPI-3 semantics allows flexibility in the time of issue of the actual transfer of RMA operation. For example, a sequence of (Lock, Accumulate and Unlock) operations may not issue a network transfer during Accumulate but defer this operation to Unlock as shown in the last entry of Figure 7.1. As a consequence, it is the combination of RMA transfer operation and synchronization method that decides the protocol used to realize it. Hence, the runtime must be considerate of this and not apply rules to transfer operations independent
of the synchronization method used. Even though window creation method can influence the protocol, it is less commonly the case.

To simplify reasoning about energy rules for the combination of transfer and synchronization, we first examine the aspects of each: 1) An RMA communication call results in either issuing a network operation immediately (after potential copy to eager intermediate buffer) or in the operation being enqueued for a deferred transfer (during a subsequent synchronization call) regardless of the synchronization method. Hence the rule of any RMA transfer operation that results in the operation being enqueued (as a entry in a list of pending operations) is to be treated as a false positive for energy savings. Incidentally, the same rule applies to RMA operations that result in the immediate issue of network transfer too as the call is commonly non-blocking from the caller’s perspective (especially with the use of RDMA). A potential copy to eager intermediate buffer (used for packing or to avoid registering) before issue is also generally short. 2) As completion of issued or pending enqueued operations are only checked for in synchronization phase, the slack that is available for reclamation is dependent on the difference in time between the actual issue and the time at which its progress is checked (Figure 7.2(a)). If this difference is larger than the threshold for a specific lower-energy gear (i.e when $s > \delta_i$ for some $i$) the scenario is to be treated as a true positive for applying $L_i$ during synchronization regardless of the communication operation performed.

There are, however, exceptions to the above generic rules. In such cases, calling an RMA data movement operation is not immediate from the caller’s perspective as it may need assistance from the remote end for progression or when direct RDMA cannot be used.
7.3.3 2-sided Realization of Synchronization Primitives

Cases where passive end needs to intervene arises from either 1) synchronization semantics or 2) due to use of operations not supported directly by hardware for true one-sidedness (eg: non-primitive or non-contiguous datatype usage, limitations in hardware support for atomic and compare-and-swap operations needed to realize Fetch-and-op or Accumulate and Get-Accumulate operations). This generally results in semantically one-sided operation being realized in a two-sided manner.

As it is the case with many MPI-2 operations, RMA payloads that require two-sided realizations typically use eager for short transfers and rendezvous for larger ones. In addition to this, however, certain control messages also need to be exchanged between interacting processes for semantic guarantees. For instance, an unlock operation needs to ensure that all RMA operations issued prior to calling it must be finished both locally and remotely.
once it returns. For this reason, most protocols tend to use acknowledgement message from the passive end (remote memory polling is less preferred owing to bad performance). Another example of using control messages is that used to initiate network transfers in lock + \text{\{RMA\}}* + unlock scenario. Before issuing the RMA operations, the origin of the RMA operations generally sends a lock request and awaits a short response packet from the target. These control messages are significant as they can often indicate skew in the application as well as delay in handshake sequences and dependencies being unfulfilled. Hence they help exploit energy saving opportunities. The general rule with such control messages (short and hence $w \approx l$) is that non-arrival of specific acknowledgements (i.e. when $s > \delta_t$) are treated as true positives for energy savings as they indicate a case where the passive end has either not arrived at a phase to service origin’s request or is busy servicing other requests during which the origin can conserve energy (Figure 7.2(b)). Specific examples for different RMA operations with different synchronizations mechanisms such as Fence and lock/unlock are discussed in Section 7.4.

For non-contiguous/non-primitive datatypes, RMA operations are realized using copy to intermediate contiguous buffers on the origin and the inverse operation (from temporary contiguous to actual offsets) occurs at target. This can have specific effects in context of using unlock which dictates completion at both origin and target. Hence, if non-contiguous datatypes are used at target, then passive side must participate (if thread support is not used) and let the origin know of its completion (of a put, for instance) through short acknowledgement control messages. The other alternative, is to forgo intermediate buffers and issue several network operations for each component of the datatype but this tends to suffer from performance degradation as opposed to the previous approach. In a similar vein, operations which involve Fetch-and-op or accumulate and get-accumulate operations on large
operands can be realized using a sequence of network operations with the exception when atomicity must be guaranteed. For all of the above operations, logGP can be appropriately used to derive rules for each individual component used to realize the whole operation in order to identify true positives and negatives for energy savings. An example is discussed in Section 7.4.

The general mechanisms for progressing RMA operations during the synchronization are as follows: 1) Fence involves using barrier synchronization for starting an epoch and reduce_scatter_block for finishing an epoch. (We avoid general active synchronization discussion due to infrequent use of the method). 2) Passive synchronization calls such as Flush and Unlock operations send packets to acquire locks from the target and wait until lock is granted to effectively start the epoch. An acknowledgement from the passive end is awaited to confirm completion of operations at the passive end which ends the epoch. These start and end points are potential sources of energy savings. When a deviation from minimum expected completion time occurs, rules are applied accordingly (Figure 7.2(b)). In both passive and active synchronization, for progressing the actual sequence of pending RMA operations, each operation of a pending list is progressed sequentially where energy rules are applied individually. Examples of typical protocols are application of energy rules are discussed in Section 7.4.

7.3.4 Implications of 2-sided MPI calls

As mentioned in Section 7.1, the derivation of rules for MPI-RMA must not perturb the performance or energy footprint of 2-sided MPI calls and vice versa. This ensures that an application that uses either of the calls benefits from both performance and energy
attributes of the runtime. As it is semantically correct to have intervening 2-sided calls in-between an RMA data transfer operation and a corresponding synchronization operation, progress status of individual outstanding RMA operations that complete are updated. In addition, enqueued operations may be initiated anew and these are monitored for progress completion and for decisions on energy savings in an upcoming synchronization phase. This ensures that performance degradation due to undertracking are minimized. Likewise, the presence of outstanding 1-sided transfer operations may have a bearing on a 2-sided operation. For instance, multiple Get operations issued prior to a Send operation or vice versa can dilate the lower bound time on the send operation if the bandwidth of the channel is reached. Here too, we leverage logGP to make energy decisions.

7.3.5 Implementation Challenges

There are challenges that a generic energy-efficient runtime must address when it is used by MPI applications for the progression of both 1-sided and 2-sided MPI calls (potentially in tandem).

7.3.5.1 Co-existence of Energy-efficient MPI 1-sided and 2-sided Protocols

Most MPI programs which use MPI-3 RMA operations overlap 1-sided data transfer calls with computation and use synchronization mechanisms at a later point to ensure either data availability or reuse of data buffers. This mode of progression generally yields itself to even pessimistic approaches which go into an energy saving execution mode on every attempt to progress network tasks in calls such MPI synchronization primitives. In other words, use of pessimistic MPI alone may suffice for energy efficiency without any notable performance degradation if the MPI application uses MPI-3 RMA calls alone. This, however, is not the case for most MPI applications as they commonly use a combination
of 2-sided and 1-sided semantics. Use of a pessimistic approach may not only cause performance degradation but also excess energy expenditure if the application predominantly uses 2-sided MPI calls and MPI-3 RMA calls sparingly. This is owing to known energy and performance drawbacks of pessimistic approach for 2-sided calls [71]. Hence, energy efficient protocols for 1-sided transfer and synchronization operations that does not affect 2-sided performance and energy are essential.

### 7.3.5.2 Guaranteeing Energy-efficiency when 1-sided and 2-sided MPI are interleaved

There may be occurrences where 2-sided MPI calls may be made between a 1-sided call and its corresponding synchronization call. Depending on the implementation, this may have the unintentional effect of progressing a pending 1-sided call at either remote or local end and the subsequent synchronization call may be effectively rendered a no-op. Due to the semantic correctness of this scenario, the energy rules derived for 1-sided calls ought to be considerate of such possible interactions with 2-sided calls. RMA rules derived independently of 2-sided calls and semantics can lead to energy inefficiency. For instance, a decision to save energy may be made in the synchronization can even when an intervening 2-sided call extinguished the possibility for it. For this reason, the rules derived for RMA must be cognizant of possible usage of 2-sided calls.

### 7.4 Case Studies for Energy-Aware MPI-3 RMA

In this section, we describe specific realizations of RMA calls and show case studies of how the generic rules for saving energy described in Section 7.3 are applied. In addition, we depict how certain unusual scenarios interact with specific implementations where energy can be saved.

171
7.4.1 RMA Data Transfer Protocols and General Progression

Most operations issued between synchronization calls can be realized in a truly one-sided manner through hardware support (like RDMA). Therefore when a particular operation is checked for completion, if the running slack $s > \delta_i$ for some $i$ then the $L_i$ is applied if $\psi_i$ is better than current lever. However operations like accumulate with large operands require two-sided protocols as hardware support for remote reduction on operands larger than the size of doubles or floats seldom exists. Consequently, the traditional rendezvous protocol is used in such cases, where, in addition to transfer of control messages such request-to-send, clear-to-send and finish control messages, there also exists a reduction component. For processing both request-to-send and reduction after payload arrival, the participation of the remote side is needed (in the absence of threads). This is one of the main sources where energy savings and possibly the primary difference from two-sided semantics where most operations are individually progressed (as decoupling of transfer and synchronization does not exist) and there is a greater expectation of remote side to arrive at the MPI context in a timely manner as an implementation artifact.

7.4.2 Active Synchronization Scenarios

Calls such as fence and the combination of post, start, complete, and wait (PSCW) constitute to active synchronization. An efficient way of realizing fence is to issue a barrier call for the first fence (ensuring that process-local load/stores on the window prior to the fence have completed). A subsequent fence call typically needs to ensure the completion of RMA transfer calls issued within the epoch. This is commonly realized using reduce_scatter_block to find the total operations for which the local window is a target for (followed by waiting for their completions). During the initial barrier, as many processes
may call fence in skewed manner due to load imbalance, there may be slack available for reclamation and likewise during the latter reduce_scatter_block due to imbalance in work between fences and 2-sided rules previously proposed ensure energy savings without degradation [71]. Figure 7.3(a) depicts potential savings that can be gained during idle time from such a scenario. Exceptions to these rules appear in the presence of info objects when NOSUCCEED and NOPRECEDE info is specified with the both being treated as false positives for energy savings if called with fence. Similar MPI-2 rules apply to PSCW scenarios where win_wait and win_complete are realized using combinations of isend and waitall and/or irecv and waitall.

As all RMA operations in-between the two fence calls are non-blocking (either because they are issued immediately or enqueued), they are all as false positives for energy savings. During the actual progress phase, each RMA operation is sequentially progressed and MPI-2 rendezvous and eager protocols energy rules from our previous work can be applied for individual calls [71].

7.4.3 Passive Synchronization Scenarios

Lock can be blocking in nature and can be realized using methods described by Mingzhe et. al [30]. But this would be treated as a false positive for most cases as it involves an issue a single remote-write operation of short message. When checking for lock access grant within lock call, the non-arrival of lock-grant acknowledgement within $\delta_i$ can be treated as a true positive for applying $L_i$. However, lock-grant access can be deferred to first RMA communication operation after that or acquiring lock can be deferred to unlock call. Deferring the majority of the work to unlock is a common way of realizing this sequence and hence unlock begins with lock-acquire-request (if exclusive-lock is used or to check
if no other process is using an exclusive-lock on target) and energy can be saved if lock- 
granted packet does not arrive in $\delta_i$ duration (Figure 7.4(a)). Next, after RMA operations 
within the lock and unlock are individually progressed. As unlock must ensure remote 
completion and an acknowledgement packet is waited upon from the remote end (in the 
absence of get operations). Therefore, rules similar to waiting for lock-granted applies. 
Figure 7.4(b) shows scenarios where passive end temporarily enters routines in MPI library 
where it services lock request and leaves MPI phase which could result in subsequent two- 
sided operations having extended durations of idleness until passive end re-enters the MPI 
library routines. As such, rules similar to those proposed for MPI-2 apply here as well.

7.4.4 Hardware Limitation Scenarios

Operations such as accumulate, get, accumulate and more generally fetch-and-op and 
compare-and-swap heavily rely on network hardware support for efficient realization. How- 
ever, most networks support fetch-and-op and compare-and-swap like operations for small 
sized operands and for simple operators such as addition and compare-and-swap. Two- 
sided protocols are often needed in order to fully support the vast list of MPI-3’s reduction 
operations (i.e without thread help) and rules discussed previously for MPI-2 semantics 
generally apply here as well.

7.4.5 Non-contiguous Datatype Usage Scenarios

Figure 7.3(b) shows how the use of non-contiguous datatypes with an operation like get 
needs explicit participation of the remote end for pack phase as many networks may not 
support scatter-gather abstractions. Energy rules for MPI-2 apply here. In the case where 
such operations are realized using multiple get operations, rules are individually applied to 
each segment.
7.4.6 Combination of synchronization methods

Passive and active synchronization methods could be used interchangeably in a program. When passive synchronization methods are followed by active synchronization methods, active synchronization methods need to make sure previous passive operations have completed at both origin and target processes. These can cause potential delays at other sites not affected by a previous non-local passive synchronization and can use the barrier or point-to-point energy rules.

7.5 Experimental Evaluation

7.5.1 Experiment Testbed

We have used TACC Stampede [4] system for performance and power measurement. Stampede is a Linux Cluster based on Dell PowerEdge server nodes, each with 2 Intel Xeon E5 (Sandy Bridge) processors. Nodes are interconnected with Mellanox FDR InfiniBand technology in a 2-level fat-tree topology. We have used MVAPICH2-2.1 as the MPI stack for implementing EAM-RMA. EAM-RMA considers polling to be the baseline power lever ($\delta_{\text{polling}} = 0, \psi_{\text{polling}} = 0$) and interrupt-driven execution mode as the sole lower-energy gear. We observed that $\gamma$ is $\approx 5\mu s$ resulting in $\delta$ to be $50\mu s$ (with an acceptable degradation $\rho = 10\%$) with interrupt-driven execution mode. We also observed that interrupt-driven execution reduces the power consumption ($\psi$) by a maximum of 66% in comparison to the polling mode. The two execution modes used for comparison are — optimistic mode (always polls) and pessimistic mode (goes into interrupt-driven execution mode at every progress attempt). MPI profiling results are collected using mpiP and it should be noted that most applications evaluated spend a large fraction in progressing 1-sided communication.
(a) Fence Finish Delay Impact on Peer Fence

(b) Fence Finish Delay with Non-Contiguous Get

Figure 7.3: Fence Scenarios
Figure 7.4: Unlocks Scenarios

(a) Late Entrance of Passive Side Into Progress

(b) Passive Side Leaves Progress Early
7.5.2 Application-kernel/Proxy-Application Evaluation

2-D Stencil Benchmark: This version of 2-D uses Put calls to realize a 5-point stencil. Problem size of 4,096 has been used for the benchmark with an energy of 5 units added in each iteration for a total of 500 iterations. The application spends majority of its communication time in Fence call that surrounds the four Put calls. Figures 7.5(a) and 7.5(b) depict the energy usage and execution time profiles. With 256 and 512 processes, EAM-RMA yields up to 40% energy savings in comparison with optimistic approach. With 512-processes, EAM-RMA yields slightly lesser savings in comparison with pessimistic approach while yielding the better performance than both the pessimistic and optimistic approaches.

![Energy Profile](a) Energy Profile

![Performance Profile](b) Performance Profile

Figure 7.5: Comparison of Optimistic, Pessimistic and EAM-RMA Runtimes with 2-D Stencil Benchmark

Graph500: Graph500 benchmark runs the breadth-first-search operation. For our experiments we run with scale and edgefactor of 20 and 16 respectively. Majority of the MPI time of this experiment is spent among Fence (progressing Accumulate operations) and
*Allreduce* calls. Figures 7.6(a) and 7.6(b) depict the energy usage and execution time profiles for process counts between 128 and 512. EAM-RMA yeilds between 31% and 46% savings with no degradation in execution time in comparison with the default optimistic MPI runtime.

![Graph500 (Energy Usage)](image1)  
(a) Energy Profile

![Graph500 (Execution Time)](image2)  
(b) Performance Profile

Figure 7.6: Comparison of Optimistic, Pessimistic and EAM-RMA Runtimes with graph500

*SCF*: This the Hartree-Fock equations for a cluster of beryllium atoms with ComEx-over-MPI backend. The input parameters for this experiment consists of 32 atoms, 64 occupied orbitals, 480 basis functions and a convergence threshold of 5.00D-04. The application spends nearly 75% of the total execution time in *Unlock* call in order to progress outstanding *Get* calls. Figures 7.7(a) and 7.7(b) depict the energy usage and execution time profiles. With 256 and 512 processes, EAM-RMA yeilds 42% and 36% savings at 11% degradation (close to permitted degradation $\rho = 10\%$) in comparison with optimistic approach.

From the above results, it can be seen that EAM-RMA is on par with energy savings and performance of pessimistic approach which is generally the best case for applications dominated by 1-sided operations. For applications that have equal time contributions from
1-sided and 2-sided calls, it is highly that the application suffers when a pessimistic approach alone is used owing to its 2-sided performance drawbacks. EAM-RMA, however, is expected to perform well on both fronts.

7.5.2.1 Related Work

Several methods to conserve energy in scientific applications have been proposed in the literature. In addition to [58], [23] and [66], Kappiah et al. use micro-operations-per-memory-load (UPM) to assign suitable frequencies for energy conservation [18]. Rong et al proposed methods of utilizing temporal patterns at the coarse granularity of application while looking at the workload characteristics using counters to conserve energy using CPUMISER [19]. Hoefler et al. have recently proposed implementing energy efficient collectives by examining the different memory, runtime and energy tradeoffs [21]. Kandalla et. al [23] and Sundriyal et. al [66] have both proposed ad hoc methods of conserving energy in specific regions within Alltoall collective by identifying idle phases.
7.5.2.2 Summary

In this chapter, we proposed Energy-Aware MPI-RMA (EAM-RMA) — A runtime which automatically saves energy without application specific knowledge, and little/no performance degradation. In this work, we analyzed commonly used RMA protocols for window creation, one-sided communication and synchronization primitives and identified potential opportunities for saving energy. Through communication modeling methods we arrived at rules for application of lower-energy without causing performance degradation in order to yield an application-oblivious runtime for automatically saving energy with a permitted user defined degradation parameter. We have also provided instances of how protocols can exploited/modified strictly for energy savings. We implement these designs using MVAPICH2 and have evaluated it with proxy-applications and kernels as Lennard Jones potential calculation and Graph500 among others. Our evaluation indicates that the EAM-RMA runtime can automatically save up to 47%, 42%, and 40% energy with Graph500, SCF, and 2-D stencil kernel respectively with 512 process at minimal performance degradation.
Impact on the Design and Use of MPI Libraries on GPU Clusters

CUDA-aware MPI libraries are the primary enablers of realizing inter-process communication of GPU data. Examples of these libraries include MVAPICH2-GPU, MVAPICH2-GDR, OpenMPI, Cray MPI, IBM Platform MPI, and SGI MPI. Prior to this dissertation, use of host memory staging to assist inter-GPU memory transfers was common. The exceptions of this include MVAPICH2-GDR, and OpenMPI, which use direct PCIe peer-to-peer-memory enabled GPUDirect-RDMA technology for low latency and low memory overhead inter-GPU transfers. These techniques, however, primarily targeted point-to-point communication and these in turn benefitted collective operations. This dissertation introduced for the first time, different means of combining existing, and upcoming network features, such as hardware multicast and collective offloading, with direct peer-to-peer-memory communication techniques. With blocking and non-blocking collective operations commonly being performance dictators in CUDA-Aware MPI applications, this has the potential to benefit a large class of applications and high-level communication and computation libraries. Furthermore, with direct peer-to-peer-memory communication being introduced for generic interfaces (eg: CAPI [63]), the techniques introduced this dissertation will influence not only InfiniBand-enabled MPI stacks but also others and will help reduce latency and improve overlap of collective operations.

Orthogonally, most CUDA-MPI applications still rely on the CPU to serve as the main orchestrator of communication and compute routines of an application. This results in low-GPU utilization, and in the wastage of many CPU cycles in satisfying dependencies between GPU compute routines and GPU-based communication. This dissertation introduced for the first time, a way to allow GPUs to execute MPI operations with minimal
CPU assistance through GPUDirect-aSync (GDS). This has a two-fold impact on application performance and energy usage. For hybrid-applications that make use of both CPU and GPU capabilities to accelerate computation, this work sets aside more CPU cycles for CPU based routines. In turn, this results in improvements in application performance. For systems that aim to reduce energy expenditure, as the technique reduces the burden placed on the CPU’s capabilities, systems with low power and low capability CPUs with high-throughput-per-watt GPUs can be built that allows the overall energy expenditure of the cluster to be reduced while allowing for overall high-throughput through GPU acceleration. In addition, the technique has a direct impact on the way task-based runtime systems are designed owing to the natural fit between GDS and scheduling systems.

7.7 Impact on the Design and Use of MPI Libraries on Xeon Phi Clusters

The algorithms and heuristics introduced in this dissertation has shown a different way of designing about collective optimizations in heterogeneous clusters with non-uniform memory costs. This is a departure from the strict use of hardware or software techniques to realize faster heterogeneity-aware collective operations. It re-emphasizes on the need to search for providing algorithmic efficiency by considering the cost differences in communication links that makeup heterogeneous clusters. Examples of such designs were shown through modified ring-allgather, recursive-doubling-allgather, bruck’s-alltoall, and pairwise-alltoall for MIC-enabled clusters. These have direct impact on existing MIC-enabled applications on clusters such as TACC-Stampede, NASA-Pleiades, etc. This also influences the way that collectives are being designed where a combination of direct memory transfer capability between many-core memory and NIC memory (eg: Intel’s Knight’s
Landing processor with onloaded NIC) and cost differences in accessing different memory regions or memory directions (expected for future NUMA systems) exist.

7.8 Impact on the Design and Use of Energy Efficient MPI Applications

With studies showing that CPU energy spent during communication routines exceed that during compute routines by an order of magnitude, the need to conserve energy during communication becomes a first-order priority. Over the past decade, in order to realize automatic energy efficient systems, researchers have primarily looked into analyzing application characteristics to predict energy usage over time or black-box approaches. This dissertation introduces for the first time, an approach that does not rely on being able to predict the running time of communication calls nor application characteristics to conserve energy with little to no performance degradation. While the work introduced looked into understanding the protocols and communication algorithms in the MPI context, this approach directly impacts many high-level communication libraries that use MPI as a communication substrate. Examples of these include libraries such as OSHMPI (PGAS), AMPI (task-parallel system), ComEx (PGAS), ARMCI (PGAS), etc. In turn, this approach will result in energy savings for a large class of applications and compute libraries that directly or indirectly use MPI without code changes and with minimal performance impact.

7.9 Software Release and Wide Acceptance

MVAPICH2 [33], is an open-source implementation of the MPI-3.1 specification over modern high-speed networks such as InfiniBand, 10GigE/iWARP, RDMA over Converged Ethernet (RoCE), and Omni-Path. This software is used by more than 2,700 organizations
in 83 countries worldwide to extract the potential of these emerging networking technologies for modern systems. As of Dec ’16, more than 406,000 downloads have taken place from this project’s site. This software is also being distributed by many vendors as part of their software distributions. The MVAPICH2 software is powering several supercomputers in the TOP500 list including the 10,649,600-core Sunway TaihuLight (ranked 1st) supercomputer, the 241,108-core Pleiades supercomputer (ranked 13th) at NASA, the 462,462-core Stampede supercomputer (ranked 17th) at TACC, and the 74,520-core Tsubame 2.5 (ranked 40th) at Tokyo Institute of Technology. MVAPICH2-GDR is a publicly available software package provides support for CUDA-Aware MPI programming model for GPU clusters. Most of the designs proposed in this dissertation for efficient MPI communication on GPU clusters are already part of the MVAPICH2-GDR software distribution. They are widely used on large scale GPU Clusters including Wilkes at the University of Cambridge, and by MeteoSwiss for weather forecasting in Switzerland. MVAPICH-EA (energy-aware) is another publicly available software package that is an implementation of MPI-3.1 specification of the MPI standard. It incorporates all the designs that have been proposed in this dissertation for automatic energy savings in MPI routines. Along with this software, members of the MVAPICH project have also made available the OSU Energy Monitoring Tool (OEMT) for comparison of energy usage across different MPI libraries and also with MVAPICH2 on applications with and without energy-awareness turned on. With over 1,350 downloads of MVAPICH2-GDR and 1,000 downloads MVAPICH2-EA in the past two years, the impact of the contributions of this dissertation is expected to be significant.
Chapter 8: Future Research Directions

In this Chapter, we present some possible future directions that can result from this dissertation.

8.1 Heterogeneity-Aware Data Movement for HPC Architectures, Deep Learning Systems, and Big Data Frameworks

This thesis looked into improving the communication performance of primarily MPI collectives on systems equipped with many-core architectures as PCIe devices. While some methods and techniques are directly applicable, questions of heterogeneity for upcoming architectures such as Intel’s Knight's Landing (KNL) and NVIDIA's DGX-1 Deep Learning System are still valid owing to the presence of heterogeneous memory entities (owing to the presence of both DRAM and multi-channel DRAM on KNL) as well as the presence of a high bandwidth network subsystem within the node (high speed NVLINK interconnect for 16 GPUs within the node) as opposed to that connecting the rest of the system (InfiniBand). These technologies present themselves as problems in communication algorithms design with non-uniform costs for specific individual paths that requires the development of heuristics and optimization methods. Lastly, runtimes which tackle Big Data problems are starting to adopt many-core architectures such as GPUs for accelerating compute phases (eg: reduction phase in MapReduce) which introduces heterogeneity
in processing power and associated in-memory data movement. Furthermore, the use of NVMe in tandem with traditional DRAM introduces heterogeneity in memory costs which potentially benefit from methods proposed in thesis when dense collective-like operations are executed.

8.2 Exploiting Control-Plane Decoupling Techniques for Collectives, One-Sided MPI operations, and for Task-Parallelism on GPU Clusters

As part of an ongoing collaboration with NVIDIA, this thesis has worked towards exploring CUDA GDS technology which is primarily targeted towards decoupling the CPU-GPU control plane. However, as GDS allows the GPU to trigger batched send operations and block CUDA stream progression until a completion queue event is generated from a target completion queue of the HCA, it is possible to realize not only 2-sided MPI_Send/MPI_Recv operations but other MPI operations as well general sequences consisting of interleaved compute and communication phases such as those in task-based runtimes. For instance, as there is an opportunity to batch together both CUDA kernels and MPI operations, it is likely profitable to realize non-blocking reduction-based collective operations. The advantage of tying together these two sets of operations is to have the ability to realize generalized reduction-based collectives which is not possible on the majority of network controllers which support collective offloading. The fact that almost any computation can be performed on the GPU allows for generalized collective offloading, with good benefits expected when reduction sizes are large due to the ability of being able to exploit the GPU’s parallelism. In addition, a natural extension to GDS-enabled stream-based MPI point-to-point operations is the design of GDS-enabled stream-based MPI collective operations. Here, the challenge is in being able to realize large scale collectives owing to
limitations on active outstanding operations as well as being able to achieve good communication progress of the collective without much assistance from the CPU. On the other hand, GDS can be used to realize 1-sided operations in a manner similar to 2-sided point-to-point operations. However, the challenges show up in two separate directions. On the one hand, semantics of 1-sided operations on CUDA streams need to be defined in manner that makes programming using them intuitive and at the same time allows for achieving good performance. On the other hand, designs to achieve good overlap without CPU assistance is not only desirable but also a prerequisite as one of the goals of moving to use of 1-sided operations is to decouple data transfer from synchronization and associated costs. Needing the CPU to constantly progress synchronization may nullify the benefits. Lastly, as task parallel runtimes gain popularity, it is useful to evaluate the benefits of using GDS based communication in tasks that perform both communication and computation based while honoring dependency directed acyclic graphs. Intuitively, GDS can be used to send data upon completion of a unit of GPU work on a stream and can also be used to wait for data to arrive or certain memory values on the GPU memory to change before initiating new work on a stream. As one of the goals of such task parallel systems is to increase concurrency and resource utilization, managing stream resources and increasing stream concurrency will remain a challenge that needs addressing.

8.3 Energy-Aware MPI (EAM) to Address Energy Usage in Big Data Frameworks

While EAM runtime proposed in this thesis addressed saving energy during MPI-2 point-to-point and collectives routines (those which are realized on top of point-to-point operations) as well as MPI-3 1-sided transfer and synchronization operations, the main
technique used behind energy saving is applicable to other runtime systems. That is, the use of communication modeling of transfer protocols and estimation of energy lever application overhead allows runtimes to make online decision on whether energy gears ought to be applied for specific communication routines. The proposed work has direct applicability in Big Data systems that are built on top of 2-sided and 1-sided MPI calls (eg: MapReduce-MPI (MR-MPI) from Sandia National Labs) but also motivates modeling map and reduce phases to be able to deduce the lower bound on the cost of these operations at different sites in order to apply energy levers. In addition, they also demand a look into energy saving mechanisms in the face of redundancy that is commonly employed in Big Data frameworks. Additional areas that motivate a further look into energy savings are that of IO phases and fault-tolerance phases in Big Data runtime systems. Unlike communication phases, these two phases exhibit the common property of being limited the lack of network and IO resources which introduces delays that are assumed to be always available in the MPI context. For instance, owing to the abundance of short message operations in the MPI context, concurrency in communication operations is assumed to be unaffected. This, however, is unlikely to be the case in Big Data frameworks where network and IO concurrency is heavily curtailed owing to contention for common resources and such patterns commonly exhibit themselves in IO phases and fault-tolerance-related phases.
Chapter 9: Conclusion and Contributions

High-Performance Computing (HPC) has increasingly resorted to the use of many-core architectures and processors with power/energy-throttling capabilities. In addition, network technologies are being developed that permits the network controller to directly access the memory of many-core processors similar to host CPU memory. While these two advancements provide the potential to increase the performance and energy profile of communication runtimes built on top of them, they require a paradigm shift from existing state-of-the-art communication designs oriented towards improving the latency, bandwidth, overlap, and energy metrics.

Most scientific applications commonly use communication substrates such as Message Passing Interface (MPI) and Partitioned Global Address Space (PGAS) in distributed environments. Most MPI communication protocols and algorithms that exist in literature have assumed uniformity in the cost of transferring data among different communication paths as well associated network processing compute costs. With heterogeneity abundant in communication costs in current and upcoming many-core systems, the uniformity assumptions of state-of-the-art communication algorithms lead to sub-optimal communication performance. This dissertation attempts to address heterogeneity challenges in communication algorithms for dense collective operations through the design of novel heterogeneity-aware collective designs. These designs are considerate of the cost differences in communication
paths and leverages delegation mechanisms, adaptations to classic algorithms, and novel heuristics to achieve upto 50-70% improvement in collectives such as MPI_Alltoall and MPI_Allgather performance at scale with both micro-benchmark experiments as well as application kernels.

In addition to changes along the communication algorithm design dimension, new communication designs that take advantage of a plethora of novel network features that allow for better communication throughput, overlap opportunities. However, they must be designed considering the idiosyncrasies of heterogeneous memories they access as well as the compute limitations where network requests originate. To this end, this dissertation also takes advantage of special-capability network subsystems that have peer-to-peer access with other PCIe devices, RDMA and multicast capabilities, offload mechanisms to design blocking broadcast operation and non-blocking collective operations that overcome the challenges posed by heterogeneous memory subsystems and yield high throughput, overlap. Specifically, this dissertation takes advantage of GPUDirect-RDMA direct memory access technology in tandem with hardware broadcast capabilities exposed by InfiniBand (IB) to design high throughput broadcast operations targeted towards streaming applications. The design achieves this through the use of IB’s scatter-gather-list (SGL) abstraction and achieves nearly 3X-4X improvement in the execution time of streaming application mimicking broadcast throughput experiment with up to 64 GPU nodes. Furthermore, the dissertation proposes for the use of non-blocking GPU collective operations for the first time by leveraging GPUDirect-RDMA and IB’s CORE-Direct collective offload technology. In doing so, the designs achieve nearly 100% overlap with collectives such as MPI_Iallgather, MPI_Ialltoall, MPI_Igather, and MPI_Iscatter at scale of 64 GPU nodes.
While the use of novel network technologies allows for faster progression of GPU collective operations, they still require the intervention of CPU to satisfy the dependencies that exist between GPU compute routines and GPU-based CUDA-Aware MPI communication routines. This results in low GPU-utilization and limits a number of cycles that the CPU can spend on its own compute routines. To this end, we propose ways of decoupling the CPU-GPU control plane and thus allow the GPU to manage its own dependencies. To achieve this, the dissertation proposes MPI extensions, and designs that make use of decoupling mechanisms exposed through NVIDIA’s GPUDirect-aSync (GDS) technology. These designs allow the GPU to issue MPI operations after the completion of CUDA kernels or block CUDA streams from launching CUDA kernels until certain MPI operations complete. The proposed designs help reduce the execution time of an application mimicking benchmark by 30% and shows promising results with micro-benchmark experiments.

Finally, communication routines have been designed solely with high-performance expectations thus far. But communication routines often result in high energy expenditure not only from the network components’ perspective but also from the CPU’s perspective owing polling policies for latency optimization. Thus, saving energy during long communication routines in an application-oblivious manner with minimal performance impact is critical. The dissertation proposes rules for automatically achieving energy savings during 2-sided and 1-sided MPI routines in an application-oblivious manner. The proposed designs achieve this through an intimate knowledge of the underlying protocols used to realize communication routines and avoid prediction methods that state-of-the-art methods rely on upon, as they lead to false positives and false negatives in the presence of system noise and irregularity in application characteristics. The proposed designs achieve between
5-40% savings in overall CPU and memory energy expenditure of a class of popular 1-sided and 2-sided applications at a scale of up to 4,096 processes.

For the widespread availability of the contributions of this dissertation, the proposed designs have largely been included into the publicly available MVAPICH2 project.
Bibliography


[41] Network-Based Computing Laboratory. MVAPICH: MPI over InfiniBand and 10GigE/iWARP. [http://mvapich.cse.ohio-state.edu/](http://mvapich.cse.ohio-state.edu/).


[56] Sreeram Potluri, Hao Wang, Devendar Bureddy, Ashish Kumar Singh, Carlos Rosales, and Dhabaleswar K Panda. Optimizing MPI Communication on Multi-GPU


