Designing High Performance and Scalable Unified Communication Runtime (UCR) for HPC and Big Data Middleware

Dissertation

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

By

Jithin Jose, M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2014

Dissertation Committee:

Dhabaleswar K. Panda, Advisor
Ponnuwamy Sadayappan
Radu Teodorescu
Karen Tomko
Abstract

The computation and communication requirements of modern HighPerformance Computing (HPC) and Big Data applications are steadily increasing. HPC scientific applications typically use Message Passing Interface (MPI) as the programming model, however, there is an increased focus on hybrid MPI+PGAS (Partitioned Global Address Space) models for emerging exascale systems. Big Data applications rely on middleware such as Hadoop (including MapReduce, HDFS, HBase, etc.) and Memcached. It is critical that these middleware be designed with high scalability and performance for next generation systems. In order to ensure that HPC and Big Data applications can continue to scale and leverage the capabilities and performance of emerging technologies, a high performance communication runtime is much needed. This thesis focuses on designing a high performance and scalable Unified Communication Runtime (UCR) for HPC and Big Data middleware.

In HPC domain, MPI has been the prevailing communication middleware for more than two decades. Even though it has been successful in developing regular and iterative applications, it can be very difficult to use MPI and maintain performance for irregular, data-driven applications. PGAS programming model presents an attractive alternative for designing such applications and provides higher productivity. It is widely believed that parts of applications can be redesigned using PGAS models - leading to hybrid MPI+PGAS applications, and improve performance. In order to fully leverage the performance benefits offered by the modern HPC systems, a unified communication runtime that offers the advantages of
both MPI and PGAS programming models is critical. We present "MVAPICH2-X" - a high performance and scalable 'Unified Communication Runtime’ that supports both MPI and PGAS programming models. This thesis also targets at redesigning applications making use of hybrid programming features, for better performance. With our hybrid MPI+PGAS design using Unified Communication Runtime, the execution time of Graph500 was reduced by 13X, compared to existing MPI based design at 16,384 processes. Similarly, the sort-rate of data intensive Out-of-Core Sort application was improved by a factor of two, using our hybrid designs.

The requirements for Big Data communication middleware are similar to those used in HPC. Both these rely on low-level communication techniques. However, the performance of current generation Big Data computing platforms, such as HBase and Memcached remains low. One of the fundamental bottlenecks in delivering high-performance on these platforms is use of the traditional BSD sockets interface and two-sided (send/recv) communication. Such communication semantics prevent the use of Remote Direct Memory Access (RDMA) and associated features of modern networking and I/O technologies. This thesis focuses on the design of a lightweight, scalable and high performance communication runtime for Big Data middleware, such as HBase and Memcached. With our UCR design, Memcached operation latencies were reduced by a factor of 12X, and HBase throughput was improved by 3X times.
To my Chachan (father), Prof. C. A. Joseph,

&

Amma (mother), Smt. Elamma Joseph
Acknowledgments

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

My family - my father, Prof. C. A. Joseph, who has always given me inspiration and complete freedom to go after my dreams; my mother, Elamma Joseph, who has given me the love and support to venture forth; my siblings, Ajay Jose and Usha Jose, who have stood by me at all times.

My wife, Renu, for her love, support and understanding. I admire and respect her for the many qualities she possesses, particularly the great courage and presence of mind she has at the most trying of times.

My advisor, Dr. Dhabaleswar K. Panda for his guidance and support throughout my doctoral program. I have been able to grow, both personally and professionally, through my association with him.

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

My friends - I am very happy to have met and become friends with Hari Subramoni, Krishna Kandalla, Sreeram Potluri, Khaled Hamidouche, Raghunath Rajachandrasekhar, Xiaoyi Lu, Jie Zhang, Akshay Venkatesh, Wasiur Rahman, Nusrat Islam, Jonathan Perkins
and Mark Arnold. This work would remain incomplete without their support and contribution. They have given me memories that I will cherish for the rest of my life.

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

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

2001-2005 .......................................................... B.Tech., Computer Science and Engineering, University of Calicut, Kerala, India
2005-2007 .......................................................... Software Engineer, Agere Systems, Bangalore, India
2007-2009 .......................................................... Software Engineer, Infineon Technologies, Bangalore, India
2009-2013 .......................................................... M.S., Computer Science and Engineering, The Ohio State University, U.S.A
2009-Present ......................................................... Ph.D., Computer Science and Engineering, The Ohio State University, U.S.A
2009-Present .......................................................... Graduate Research Associate, The Ohio State University, U.S.A
2012 ................................................................. Summer Research Intern, IBM Research, U.S.A.
2013 ................................................................. Summer Research Intern, IBM Research, U.S.A.

Publications


Jithin Jose, Sreeram Potluri, Hari Subramonoi, Xiaoyi Lu and Dhabaleswar K. Panda, Karl Schulz, and Hari Sundar, *Designing Scalable Out-of-core Sorting with Hybrid MPI+OpenSHMEM Programming Models*, (under review)


Jithin Jose, Krishna Kandalla, Miao Luo, and Dhabaleswar K. Panda, Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation, *International Conference on Parallel Processing (ICPP)*, September 2012


Jithin Jose, Khaled Hamidouche, Jie Zhang, Akshay Venkatesh, and Dhabaleswar K. Panda, Optimizing Collective Communication in UPC, *International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, held in conjunction with *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2014


Mingzhe Li, Xiaoyi Lu, Sreeram Potluri, Khaled Hamidouche, Jithin Jose, Karen Tomko and Dhabaleswar K. Panda, Scalable Graph500 Design with MPI-3 RMA, IEEE International Conference on Cluster Computing (CLUSTER), September 2014

Hari Subramoni, Krishna Kandalla, Jithin Jose, Karen Tomko, Karl Schulz, Dmitry Pekurovsky and Dhabaleswar K. Panda, Designing Topology Aware Communication Schedules for All-to-All Operations in Large InfiniBand Clusters, International Conference on Parallel Processing (ICPP), September 2014

Jie Zhang, Xiaoyi Lu, Jithin Jose, Rong Shi, and Dhabaleswar K. Panda, Can Inter-VM Shmem Benefit MPI Applications on SR-IOV based Virtualized InfiniBand Clusters?, International Conference EuroPar Parallel Processing (EuroPar), August 2014


Mingze Li, Sreeram Potluri, Khaled Hamidouche, Jithin Jose and Dhabaleswar K. Panda, Efficient and Truly Passive MPI-3 RMA Using InfiniBand Atomics, EuroMPI, September 2013


**Fields of Study**

Major Field: Computer Science and Engineering
Table of Contents

Abstract .......................................................... ii
Dedication ......................................................... iv
Acknowledgments .................................................. v
Vita ................................................................. vii
List of Tables ...................................................... xvii
List of Figures ...................................................... xviii

1. Introduction .................................................... 1
   1.1 Problem Statement ........................................ 5
   1.2 Research Framework ........................................ 7

2. Background .................................................... 13
   2.1 InfiniBand .................................................. 13
      2.1.1 Communication Model and Transports .............. 13
      2.1.2 Communication Semantics ........................... 14
   2.2 Programming Models for High Performance Computing .... 15
      2.2.1 Message Passing Interface (MPI) .................... 15
      2.2.2 PGAS Models .......................................... 17
         2.2.2.1 OpenSHMEM ....................................... 17
         2.2.2.2 Unified Parallel C (UPC) ......................... 18
   2.3 Graph500 Benchmark ....................................... 18
   2.4 Out-of-core Sort Application ............................. 19
   2.5 Big Data Middleware ....................................... 21
      2.5.1 Memcached ........................................... 21
2.5.2 HBase ................................................. 22
2.6 Big Data Middleware Benchmarks .................................. 23
  2.6.1 Apache Olio ........................................... 23
  2.6.2 Yahoo Cloud Serving Benchmark (YCSB) ...................... 23
2.7 Communication on Clusters with Intel MIC based Coprocessors .... 24

3. MVAPICH2-X: Unified Communication Runtime for MPI and PGAS .... 27
  3.1 UCR: Design Requirements and Solutions for UPC Runtime ........... 27
    3.1.1 Design Requirements .................................... 27
    3.1.2 Approach ............................................ 28
    3.1.3 Internal Implementation Details .......................... 29
    3.1.4 Experimental Results ................................... 33
      3.1.4.1 Experimental Platform ................................. 34
      3.1.4.2 Microbenchmark Level Evaluation ...................... 34
      3.1.4.3 Scalability Evaluation ................................. 38
      3.1.4.4 MPI Application Evaluation ............................ 39
      3.1.4.5 UPC Application Level Evaluation ...................... 40
  3.2 UPC Queues for Scalable Graph Traversals .......................... 43
    3.2.1 Design Requirements .................................... 43
    3.2.2 UPC Queue Operations ................................... 45
    3.2.3 Design and Implementation ................................ 47
    3.2.4 Experimental Results ................................... 51
      3.2.4.1 Experimental Platform ................................. 51
      3.2.4.2 Micro-benchmark Performance .......................... 52
      3.2.4.3 Graph500 Benchmark Performance ...................... 54
      3.2.4.4 Unbalanced Tree Search Benchmark Performance .......... 55
  3.3 Optimizing Collective Communication in UPC ........................ 56
    3.3.1 Design and Implementation ................................ 57
    3.3.2 Experimental Evaluation ................................ 60
      3.3.2.1 Experiment Setup ................................... 60
      3.3.2.2 MicroBenchmark Evaluations .......................... 61
      3.3.2.3 UPC Application Evaluation ............................ 69
  3.4 UCR Enhancements for OpenSHMEM Runtime ........................... 71
    3.4.1 OpenSHMEM Design over UCR ................................ 72
    3.4.2 Hybrid (OpenSHMEM + MPI) Programming Support ............... 75
    3.4.3 Experimental Evaluation ................................ 76
      3.4.3.1 Experiment Setup ................................... 76
      3.4.3.2 Microbenchmark Evaluation ............................ 77
      3.4.3.3 OpenSHMEM Application Evaluation ...................... 83
      3.4.3.4 Hybrid Application Evaluation ........................ 84
      3.4.3.5 Memory Footprint Evaluation of Hybrid Applications ..... 87
3.5 Optimizing Collective Communication in OpenSHMEM
   3.5.1 Design Challenges ........................................ 90
   3.5.2 Detailed Design ........................................... 92
   3.5.3 Experimental Evaluation ................................. 94
      3.5.3.1 Experiment Setup ................................ 94
      3.5.3.2 MicroBenchmark Evaluations ...................... 95
      3.5.3.3 OpenSHMEM Application Evaluation .............. 98

3.6 Summary ...................................................... 100

4. Hybrid MPI+OpenSHMEM Design of Graph500 .................. 103
   4.1 Bottlenecks in Graph500 MPI Version ...................... 103
   4.2 Design and Implementation ................................ 106
      4.2.1 Design Challenges .................................... 106
      4.2.2 Detailed Design ....................................... 107
         4.2.2.1 Communication using One-sided Routines .... 108
         4.2.2.2 Co-ordination using Fetch-add Atomic Operation 108
         4.2.2.3 Buffer structure for Computation-communication Overlap 109
         4.2.2.4 Load Balancing .................................... 109
   4.3 Experimental Evaluation .................................... 112
      4.3.1 Experiment Setup ..................................... 112
      4.3.2 Performance Evaluation ............................... 114
      4.3.3 Evaluation of Load Balancing ....................... 116
      4.3.4 Scalability Analysis .................................. 118
   4.4 Summary ..................................................... 120

5. Hybrid MPI+OpenSHMEM Design of Out-of-Core Sort ........ 121
   5.1 Existing MPI Based Design and its Limitations ............ 121
      5.1.1 Overview of Existing MPI-based Design ............ 121
      5.1.2 Overheads/Bottlenecks in Current Design .......... 124
   5.2 Design and Implementation ................................ 126
      5.2.1 Design Challenges .................................... 126
      5.2.2 Proposed Hybrid Design Architecture ............. 127
      5.2.3 Detailed Design ....................................... 128
         5.2.3.1 Global View of Memory and Management .... 128
         5.2.3.2 Read Group ........................................ 129
         5.2.3.3 Sort Group ........................................ 130
         5.2.3.4 Data Delivery ..................................... 130
   5.3 Performance Evaluation ..................................... 132
      5.3.1 Experimental Setup .................................. 133
      5.3.2 Evaluation of Different Phases of Sort Operation 134
7.3.1 Enhancing UCR to support Hybrid Transports ............... 180
  7.3.1.1 Reliability and Flow Control ....................... 180
  7.3.1.2 Large Message Transfers .......................... 180
  7.3.1.3 Hybrid Connection Management ..................... 181
  7.3.1.4 Communication Performance ....................... 181
7.3.2 Memcached based on UCR-UD .......................... 181
  7.3.2.1 Performance Evaluation .......................... 182
    7.3.2.1.1 Experimental Setup .......................... 182
    7.3.2.1.2 Microbenchmark Analysis ..................... 183
    7.3.2.1.3 Performance of Memcached Set/Get operations 183
    7.3.2.1.4 Transaction Throughput ....................... 186
    7.3.2.1.5 Scalability Analysis ......................... 188
    7.3.2.1.6 Apache Olio Benchmark Results .............. 189
    7.3.2.1.7 Results based on Real Application Workloads . 191
7.4 Summary ............................................ 193

8. High Performance RDMA-based design of HBase .................. 195
  8.1 Detailed Design ...................................... 195
    8.1.1 Socket-based HBase Communication Flow ............. 195
    8.1.2 Hybrid Communication Framework to Support High Performance Networks .......................... 197
    8.1.3 Communication Flow over InfiniBand via UCR library . 197
    8.1.4 Connection Management ............................ 198
    8.1.5 Communication Buffer Management .................. 200
  8.2 Performance Evaluation ............................... 200
    8.2.1 Experimental Setup ............................... 201
    8.2.2 Micro-benchmark Evaluations ...................... 203
    8.2.3 Detailed Profiling Analysis ..................... 205
    8.2.4 YCSB (Single server and Multiple clients) ........ 207
    8.2.5 YCSB: Multi-servers and Multi-clients ............ 207
  8.3 Summary ............................................ 212

9. Impact on the HPC and Big Data Communities .................... 213
  9.1 Impact on the Design and Use of MPI and PGAS Libraries .... 213
  9.2 Software Release and Wide Acceptance .................... 213

10. Future Research Directions ................................ 215
  10.1 Hierarchical Hybrid MPI+PGAS Models ................... 215
  10.2 Exploring Streaming Semantics in UCR ................... 216

xv
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Time Spent in MPI Routines for MPI_Simple Implementation</td>
<td>104</td>
</tr>
<tr>
<td>5.1</td>
<td>Percentage of Time Spent during Read Stage</td>
<td>124</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Research Framework</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Overview of end-to-end Sort process</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Memcached Architecture and Network Model</td>
<td>21</td>
</tr>
<tr>
<td>2.3 HBase Architecture and Network Model</td>
<td>22</td>
</tr>
<tr>
<td>2.4 Communication Channels among MICs and Hosts</td>
<td>25</td>
</tr>
<tr>
<td>3.1 An overview of various communication stack options available for MPI and UPC applications and the proposed unified runtime</td>
<td>28</td>
</tr>
<tr>
<td>3.2 Memput performance</td>
<td>35</td>
</tr>
<tr>
<td>3.3 Memget performance</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Berkeley UPC (signal) memput performance</td>
<td>36</td>
</tr>
<tr>
<td>3.5 Scalability Analysis and MPI Performance Evaluation</td>
<td>39</td>
</tr>
<tr>
<td>3.6 Performance of UPC NAS Benchmarks</td>
<td>42</td>
</tr>
<tr>
<td>3.7 Implementation of Queues in UPC Runtime</td>
<td>48</td>
</tr>
<tr>
<td>3.8 Enqueue/Dequeue operation in UPC Queue</td>
<td>50</td>
</tr>
<tr>
<td>3.9 Performance comparison of enqueue-dequeue operation in different implementations of queues</td>
<td>53</td>
</tr>
</tbody>
</table>
3.10 Performance comparison of enqueue-dequeue operation in different implementations of queues, for a 128 byte queue item, on varying system sizes . 54

3.11 Application Performance ...................... 55

3.12 Overview of Berkeley UPC Communication Stack .................. 58

3.13 Proposed Design of UPC Collectives in MVAPICH2-X .............. 59

3.14 Performance of upc_all_broadcast ..................... 63

3.15 Performance of upc_all_scatter ....................... 64

3.16 Performance of upc_all_gather ....................... 66

3.17 Performance of upc_all_gather_all ................... 67

3.18 Performance of upc_all_exchange .................... 68

3.19 Performance of UPC Applications ..................... 70

3.20 Latency and Bandwidth Comparison of shmem_getmem ............. 78

3.21 Latency and Bandwidth Comparison of shmem_putmem ............. 78

3.22 Performance of Atomic Memory Routines .................. 80

3.23 Performance of Collective Operations .................. 82

3.24 Performance of OpenSHMEM Applications ................ 85

3.25 Performance of Hybrid (MPI+OpenSHMEM) Applications ........... 87

3.26 Network Resource Consumption for Hybrid Program ............ 88

3.27 Proposed Design of OpenSHMEM Collectives ................. 93

3.28 Collect Performance Comparison (All Processes) (a) 128 Processes, (b) 256 Processes, and (c) 512 Processes ..................... 97
5.8 Overlap Evaluation .............................................. 136
5.9 Output File Size Distribution ................................. 137
5.10 Overall Execution Time of Sort ............................. 138
5.11 Strong Scalability Evaluation ............................... 139
6.1 Proposed Application Interfaces for Team ................... 143
6.2 Team-based non-uniform memory allocation ............... 144
6.3 Example of shmalloc_team usage ......................... 145
6.4 Structure of Address Handle .................................. 146
6.5 Proxy based design for OpenSHMEM communication operations ..... 149
6.6 Intra-node Put Latency ........................................ 154
6.7 Inter-node Put Latency ....................................... 155
6.8 Intra-node Get Latency ....................................... 156
6.9 Inter-node Get Latency ....................................... 156
6.10 Broadcast Latency ............................................. 158
6.11 Reduce Latency ............................................... 158
6.12 Collect Latency ............................................... 158
6.13 Barrier Latency ............................................... 159
6.14 Performance Evaluations using Graph500 .................. 160
7.1 Active Messaging in UCR ..................................... 164
7.2 Network Transactions for Memcached Set and Get Operations using UCR 168
7.3 Latency of Set and Get Operations on Intel Clovertown Cluster .............. 173
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>Latency of <em>Get</em> and <em>Set</em> Operations on Intel Westmere Cluster</td>
<td>176</td>
</tr>
<tr>
<td>7.5</td>
<td>Latency of Small Messages for Non-Interleaved (Set 10% Get 90%) and</td>
<td>177</td>
</tr>
<tr>
<td></td>
<td>Interleaved (Set 50% Get 50%) Operations on Intel Clovertown and Intel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Westmere Clusters</td>
<td></td>
</tr>
<tr>
<td>7.6</td>
<td>Number of Transactions per Second for <em>Get</em> Operation on Intel Clovertown</td>
<td>179</td>
</tr>
<tr>
<td></td>
<td>and Intel Westmere Clusters</td>
<td></td>
</tr>
<tr>
<td>7.7</td>
<td>Latency of Memcached Operations - Cluster A</td>
<td>184</td>
</tr>
<tr>
<td>7.8</td>
<td>Latency of Memcached Operations - Cluster B</td>
<td>187</td>
</tr>
<tr>
<td>7.9</td>
<td>Scaling Memcached - Transaction Throughput and Memory Footprint</td>
<td>189</td>
</tr>
<tr>
<td>7.10</td>
<td>Apache Olio Benchmark Results</td>
<td>190</td>
</tr>
<tr>
<td>7.11</td>
<td>Real Application Workload Results</td>
<td>192</td>
</tr>
<tr>
<td>8.1</td>
<td>Architecture and Workflow of HBase Design with Hybrid Approach</td>
<td>196</td>
</tr>
<tr>
<td>8.2</td>
<td>HBase Put/Get Latency (Server, Single Client)</td>
<td>202</td>
</tr>
<tr>
<td>8.3</td>
<td>Get/Put Query Processing Time Breakdown (on Cluster A, 1 KB Record</td>
<td>205</td>
</tr>
<tr>
<td></td>
<td>Size)</td>
<td></td>
</tr>
<tr>
<td>8.4</td>
<td>Single Server, Multiple Clients, HBase <em>Get</em> on Cluster A, 1 KB Message</td>
<td>206</td>
</tr>
<tr>
<td></td>
<td>Size.</td>
<td></td>
</tr>
<tr>
<td>8.5</td>
<td>Multi-Servers Multi-Clients, Read-only or Write-only Workload on Cluster</td>
<td>208</td>
</tr>
<tr>
<td></td>
<td>A (1 thread/node, 1 KB message)</td>
<td></td>
</tr>
<tr>
<td>8.6</td>
<td>Multi-Servers Multi-Clients, Read-Write Workload on Cluster A (1 thread/node, 1 KB message)</td>
<td>208</td>
</tr>
<tr>
<td>8.7</td>
<td>Multi-Servers Multi-Clients, Read-only or Write-only Workload on Cluster</td>
<td>211</td>
</tr>
<tr>
<td></td>
<td>A (8 threads/node, 1 KB message)</td>
<td></td>
</tr>
<tr>
<td>8.8</td>
<td>Multi-Servers Multi-Clients, Read-Write Workload on Cluster A (8 threads/node, 1 KB message)</td>
<td>212</td>
</tr>
</tbody>
</table>

xxii
Chapter 1: Introduction

Modern high-end computing (HEC) systems allow scientists and engineers to tackle various grand challenge problems, such as, astrophysics, earthquake analysis, weather prediction, nanoscience modeling, multi-physics modeling, biological computations, computational fluid dynamics, etc. The ‘Big Data’ analytics is fundamentally changing the way decisions are made in a wide range of domains including health care, biomedical research, internet search, finance and business informatics, etc. These applications have a voracious need for computing power and this increasing demand is met by modern high-end computing systems that can offer sustained peta-flop performance. The communication requirements of these applications are also increasing and this has driven the evolution of modern networking technologies.

The design and deployment of next generation ultra-scale systems is fueled by the increasing use of multi-core environments. The use of many-core architectures in these systems is helping to accelerate computation. The emergence of commodity networking technologies like InfiniBand [5] with the Remote DMA (RDMA) feature is also fueling this growth and allowing petascale systems to be designed with commodity cluster configurations at relatively modest costs. Based on the June 2014 TOP500 [10] ranking, there are 222 clusters (nearly 44%) in the Top 500. The latest InfiniBand adapters from Mellanox, Connect-IB [54] offer dual Fourteen Data Rate (FDR) with 100Gbps to/from a node, along
with many new features to improve the performance and scalability of HPC and Big Data applications. However, can real scientific and big data applications, which may have irregular computation and communication patterns, exploit the communication and computation power offered by emerging high performance networks and many-core architectures? How can the requirements of next generation applications be satisfied with the programming models and runtimes for next generation petascale/exascale systems? These are some of the most challenging questions facing the parallel programming models and application development communities. Unless these questions are resolved, it is unlikely that scientific and big data applications will be able to answer grand challenge problems.

MPI has been the prevailing communication middleware between scientific applications and HPC systems for more than two decades. MPI has been widely adopted across various scientific domains and is very successful in implementing regular, iterative parallel algorithms with well defined communication patterns. However, as the computing, networking, and storage technologies continue to evolve, there are many challenges to harness these capabilities at the MPI layer. The latest revision of the MPI Standard - MPI-3.0 [56] offers many new modern features to improve the performance of parallel applications.

The Partitioned Global Address Space (PGAS) programming models present an alternative approach compared to Message Passing (MPI). PGAS models such as Unified Parallel C (UPC) [2] and Co-Array Fortran (CAF) [4] use language-based approaches and have attracted much attention the last decade. Library-based PGAS models like OpenSHMEM [63] are also gaining momentum in the exascale community to deliver performance and productivity. In PGAS model, data can be stored in global arrays and manipulated by individual compute “threads.” This model shows promise for expressing algorithms that have irregular computation and communication patterns. Real-world applications that
are data-driven often exhibit the following characteristics: a) there is no obvious way to partition the data in a way that computation is load-balanced; and b) over the duration of computation, data may migrate (often according to spatial locality), and there is no obvious or easy method to re-load-balance data. These problems are often very challenging for individual application developers to solve in MPI. Further complicating matters is the fact that real-world applications depend on many parallel numerical libraries which have been written in MPI, the hitherto de-facto parallel programming model, and it requires significant effort to re-write these in PGAS model.

The hybrid MPI+PGAS programming models enable application scientists to take advantage of both MPI and PGAS models, without having to rewrite the entire application. Sub-kernels with regular and iterative characteristics can be implemented using MPI whereas irregular and data intensive parts can be implemented using PGAS. The Exascale roadmap identifies hybrid model as the “practical” way of programming Exascale systems [25]. However, using separate runtimes for MPI and PGAS models will result in poor performance and can result in deadlock between runtimes [3]. Coercing one programming model on another hammers the performance, as there is a fundamental difference in the communication semantics between the two programming models. This leads to the following broad challenges: **Can a unified runtime be designed to support hybrid programming models? Can the unified runtime for hybrid programming model provide benefits that are greater than the sum of its parts?**

The communication characteristics of big-data middleware such as Memcached [7] and HBase [14] are also similar to that in HPC applications, as they also rely on low-level techniques that are similar to those used in the HPC middleware. Memcached is a memory caching layer, which caches the results of previous database queries. In an environment
dominated by read operations, such caching can prevent expensive database queries in the critical path. Memcached is termed as the ‘pillar’ of Web 2.0. HBase – the Hadoop database – is an open-source implementation of Google’s BigTable [20]. The goal of HBase is to provide random, real time read/write access to data in the order of billions of rows and millions of columns. HBase can be used as the source and sink for MapReduce jobs. Both HBase and Memcached are widely used by many Internet companies (e.g. Facebook, Twitter, and Yahoo!) because of its open source model and high scalability. Even though the architectures of Memcached and HBase are designed for high performance and scalability, the performance observed with these middleware is lower. One of the fundamental bottlenecks in delivering high-performance on these middleware is use of the traditional BSD sockets interface with two-sided (send/recv) communication. Such communication semantics prevent the use of RDMA and associated features of modern networking technologies. Moreover, there exists a fundamental difference in the ‘memory-object’ semantics of Memcached and HBase, as compared to the ‘byte-stream’ semantics of sockets. This results in multiple memory-copies and hammers the performance. On the other hand, the memory-object semantics matches perfectly with the RDMA communication model offered by high performance networks. This leads to the broad challenges: **Can a unified runtime be designed to provide RDMA-based communication for big data middleware? Can the middleware be re-designed using the unified runtime to provide efficient communication?**

The proposed research takes on these fundamental challenges to achieve the following objectives: 1) Design a unified runtime to support hybrid programming models and big data middleware, 2) Explore hybrid MPI+PGAS programming models for expressing irregular and data intensive applications, and 3) Re-design a set of driving petascale applications
and big data middleware to use this runtime and associated hybrid programming models to demonstrate performance, scalability and productivity.

1.1 Problem Statement

As indicated above, hybrid MPI+PGAS programming models enable application scientists to take advantage of both MPI and PGAS models, without having to rewrite the entire application. The Exascale road-map denotes hybrid programming as a ‘practical’ way for programming exascale systems [25]. However, there is no runtime that can support these models and deliver the best performance and scalability for MPI+PGAS hybrid programming paradigms. Individual programming models (such as MPI, UPC, OpenSHMEM, OpenMP, etc.) have been designed and tuned for homogeneous architectures for many years. The GASNet project has been trying to standardize PGAS runtimes. It provides support for MPI+UPC applications by implementing UPC over MPI. However, this does not provide the best performance due to various limitations in the MPI standard which prevent it from being used as an underlying layer for UPC [18]. Implementing one model over the other often results in poor performance because of the mismatch in semantics of both models. Another way is to have multiple runtimes, which leads to waste of network resources. It also imposes the additional requirement that application shall ensure progress of both runtimes for avoiding possible deadlocks [3]. Instead, we believe that the best path forward is to have a true unification between MPI, and PGAS runtime libraries. The absence of such a high performance and scalable unified runtime is preventing application developers and the HEC community’s ability to fully leverage the benefits of these programming paradigms on modern architectures. Similarly, the socket based communication schemes in big-data middleware prevents these from using advanced features
such as RDMA. Also, the semantics mismatch between the stream model of sockets and memory model of big-data middleware blocks the use of RDMA-aware designs. We believe the unified runtime can support both HPC and big data communication requirements. To summarize, the unified communication runtime should support the following with high performance and scalability – pure models (MPI, OpenSHMEM, UPC), hybrid models (MPI+UPC, MPI+OpenSHMEM), and data-center middleware such as Memcached and HBase.

In this thesis, we target these issues for supporting hybrid programming models and enable high performance RDMA-aware communication for big data middleware. To summarize, it addresses the following broad challenges:

1. What are the requirements of using hybrid programming models for a set of applications on modern system architectures?
2. What features and mechanisms are needed in a unified runtime to satisfy the requirements?
3. How can the unified runtime be designed and implemented by taking care of systems and interconnect characteristics of modern architectures?
4. How can the candidate applications be redesigned to take advantage of proposed unified runtime and hybrid programming models?
5. What kind of benefits (in terms of performance, scalability, and productivity) can be achieved by the proposed designs?
6. Can we extend the unified runtime to support big data middleware and enable RDMA-aware communication?
7. What kind of benefits (in terms of performance and scalability) can be achieved for the big data middleware by the proposed designs?

1.2 Research Framework

Figure 1.1 depicts the research framework that we propose to address the challenges highlighted above. We discuss how we use the framework to addresses each of the challenges in detail.

Figure 1.1: Research Framework

1. What are the requirements of using hybrid programming models for a set of applications on modern system architectures?

With the data requirements doubling every year, data intensive applications are increasing linearly. Such applications usually have highly irregular communication
characteristics. Graph500 [78] benchmark is a representative benchmark for such applications with data intensive and irregular communication characteristics. It is used to rank supercomputers, based on such workloads. Out-of-Core Sort is used for sorting very huge data sets, which cannot be stored in the combined main memories of all the participating nodes. In this thesis, we chose Graph500 and Out-of-Core Sort as the representative applications on modern system architectures. Further, we also consider NAS Parallel Benchmarks [8], which represents scientific application characteristics on current generation systems. We choose Memcached and HBase as representative big-data middleware, because of their data-intensive nature.

The variable and irregular communication characteristics require application designers to use MPI and PGAS models interchangeably. The proposed unified runtime enables application developers to develop applications in a deadlock-free manner, using both MPI and PGAS, without having to worry about the intricacies of multiple runtimes. In the following sections, we describe how Graph500, Out-of-Core Sort, Memcached, and HBase are redesigned using the unified communication runtime.

2. What features and mechanisms are needed in a unified runtime to satisfy the requirements?

In this thesis, we profile and analyze the characteristics of the representative applications, and garner the requirements for unified runtime. Based on these, we propose the unified communication runtime as a light-weight, and low-memory footprint runtime, which allows deadlock free communication progress for both MPI and PGAS models. Network features such as Remote Direct Memory Access (RDMA), atomic operations, high performance collective operations are highly desired for both
MPI and PGAS. For one-sided operations, InfiniBand hardware requires the source and destination addresses to be registered with Network Interface Controller (NIC). Memory register operations are heavy operations. In this thesis, we propose a high performance runtime and also propose mechanisms such as local and remote registration caches, to alleviate such limitations in a seamless manner, transparent to the application layer.

3. How can the unified runtime be designed and implemented by taking care of systems and interconnect characteristics of modern architectures?

This thesis presents the design and implementation of unified communication runtime, which is based on the application characteristics and runtime feature requirements gathered. The main design blocks of the proposed unified runtime are active messages, direct memory update operations, atomic memory updates, light-weight collective operations, and hybrid transport protocols. Active messages implement several utility functions and synchronization operations. Direct memory and atomic updates implement one-sided put/get operations, and atomic operations. The unified runtime proposes designs for high performance and efficient collective routines which can be used for both MPI and PGAS models. As discussed earlier, InfiniBand communication requires local and remote memory to be registered with NIC. The proposed runtime overcomes this requirement by keeping local and remote registration caches. The detailed design and implementation details are presented in following sections. As part of future work, we plan to re-design the collective operations for PGAS models, based on the heavily researched and scalable collective communication schemes defined for MPI.
4. How can the candidate applications be redesigned to take advantage of proposed unified runtime and hybrid programming models?

Hybrid MPI+PGAS programming model changes the way in which communication is designed in applications. It offers the best properties of both MPI and PGAS models for application design. Application scientists can choose the model for sub-kernels based on the communication characteristics. There has been some application-level studies [24, 70], utilizing both MPI and PGAS models. However, these applications use separate runtimes for MPI and PGAS. In this proposal, we study the challenges in redesigning Graph500 and Out-of-Core Sort applications, which have highly irregular and data-intensive communication characteristics, using hybrid models. We propose efficient hybrid designs for both the applications. In these designs, we keep sub-kernels which are best suited for MPI as it is, and redesign other sub-kernels using PGAS model. The hybrid designs exhibit significant performance improvements. This thesis also studies the performance characteristics of hybrid applications with single unified runtime and with separate runtimes.

5. What kind of benefits (in terms of performance, scalability, and productivity) can be achieved by the proposed designs?

Traditionally, applications are written in either using pure MPI or pure PGAS model. Since PGAS models present an easier way to express irregular parallelism, application scientists can redesign sub-kernels using PGAS models, resulting in hybrid applications. However, if separate runtimes are used for MPI and PGAS, application scientists has to take special steps to ensure progress of both runtimes, for avoiding deadlocks between runtimes. Deadlocks causing from the interaction of multiple
runtimes are hard to detect. Application scientists need to be aware of the underlying runtime designs to solve such issues. Often additional synchronization routines need to be added, which significantly impact performance. On the other hand, unified runtime enables hybrid programming, without having application scientists to worry about underlying runtime issues. Further, the separate runtime approach requires network resources for both runtimes, resulting in higher memory requirements for the hybrid application. In this thesis, we study the performance, memory requirements and scalability characteristics of hybrid designs. We present the hybrid designs of Graph500 and Out-of-Core sort, and study the characteristics.

6. Can we extend the unified runtime to support big data middleware and enable RDMA-aware communication?

The proposed unified communication runtime aims to unify the communication runtime requirements of scientific parallel programming models, such as MPI and Partitioned Global Address Space (PGAS) along with those of data-center middleware, such as Memcached and HBase. Our approach is to re-use the best communication runtime designs across these domains. The requirements imposed by these disparate models are quite challenging. For example, in the parallel computing domain, there is a notion of a parallel “job.” Processes belonging to a job may communicate with others using a rank. This is quite different from the data-center middleware model, where clients maintain a list, or pool, of available servers. Based on the data they want to access, keys are assigned to the data and a server is chosen using a hash function. Another important distinction is that of fault tolerance. In MPI or PGAS, when a process belonging to a job unexpectedly fails, the entire job fails. However, in the data-center domain, failure of one Memcached server or client must be tolerated. In
this thesis, we enhance the UCR proposed for scientific parallel programming models by enhancing the active messages, and supporting the connection establishment and communication mode such that clients can come and go anytime. The proposed runtime enables RDMA-aware communication for data-center middleware.

7. What kind of benefits (in terms of performance and scalability) can be achieved for the big data middleware by the proposed designs?

Most of the data-center middleware are data and network intensive in nature. Memcached – a memory caching layer – aggregates the memory from multiple servers and acts as a cache to the back-end database. Clients find the server based on a special function on the lookup key. Even though the model is scalable, the socket based communication model hammers the performance. Moreover there exists a fundamental difference between the byte-stream oriented sockets model and the memory-object model of data-center middleware. Similar bottlenecks exist for HBase, which is the Hadoop database. In this thesis, we extend the UCR to support data-center middleware. We propose the high performance RDMA-aware designs of Memcached using UCR. As part of future work, we plan to re-design HBase and also consider hybrid transport protocols for higher scalability.
Chapter 2: Background

2.1 InfiniBand

InfiniBand (IB) is an industry standard switched fabric that is designed for interconnecting compute and I/O nodes in High-End Computing clusters [5]. It has emerged as the most-used internal systems interconnect in the Top 500 list of supercomputers. The list release in June 2013 reveals that more than 40% of the systems use IB.

2.1.1 Communication Model and Transports

Connection establishment and communication in IB is done using a set of primitives called Verbs. IB uses a queue based model. A process can queue up a set of instructions that the hardware executes. This facility is referred to as a Work Queue (WQ). Work queues are always created in pairs, called a Queue Pair (QP), one for send operations and one for receive operations. In general, the send work queue holds instructions that cause data to be transferred from one process’s memory to another process, and the receive work queue holds instructions about where to place data that is received. The completion of Work Queue Entries (WQEs) is reported through Completion Queues (CQ). Memory involved in communication through IB should be registered with the IB network adapter. Registration is done using an IB verbs call which pins the corresponding pages in memory and returns
local and remote registration keys (lkey and rkey). The keys are used in communication operations as described in the following section. InfiniBand offers four transport modes - Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC) and Unreliable Datagram (UD). RC is a connection-oriented service and it requires a distinct QP per communicating peer. It provides RDMA capability, atomic operations and reliable service. UD is a connectionless and unreliable transport. A single UD QP can communicate with any number of other UD QPs. However, UD does not offer RDMA, reliability and message ordering. Moreover, messages larger than an MTU size (2 KB in current Mellanox hardware) have to be segmented and sent in MTU size chunks.

2.1.2 Communication Semantics

InfiniBand supports two types of communication semantics in RC transport: channel and memory semantics. In channel semantics, both the sender and receiver have to be involved to transfer data between them. The sender has to post a send work request entry (WQE) which is matched with a receive work request posted by the receiver. The buffer and the lkey are specified with the request. It is to be noted that the receive work request needs to be posted before the data transfer gets initiated at the sender. The receive buffer size should be equal or greater than the send buffer size. This restriction is prevalent in most high-performance networks like Myrinet [17], Quadrics [9] and others. This allows for zero-copy transfers but requires strict synchronization between the two processes. Higher level libraries avoid this synchronization by pre-posting receive requests with staging buffers. The data is copied into the actual receive buffer when the receiver posts the receive request. This allows the send request to proceed as soon as it is posted. There exists a trade-off between synchronization costs and additional copies. In memory semantics, Remote Direct
Memory Access (RDMA) operations are used instead of send/receive operations. These RDMA operations are one-sided and do not require software involvement at the target. The remote host does not have to issue any work request for the data transfer. The send work request includes address and lkey of the source buffer and address and rkey of the target buffer. Both RDMA Write (write to remote memory location) and RDMA Read (read from remote memory location) are supported in InfiniBand.

### 2.2 Programming Models for High Performance Computing

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

#### 2.2.1 Message Passing Interface (MPI)

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

**The MVAPICH2 MPI Library:** MVAPICH2 [57, 65], is an open-source implementation of the MPI-3 specification over modern high-speed networks such as InfiniBand,
MVAPICH2 delivers best performance, scalability and fault tolerance for high-end computing systems and servers using InfiniBand, 10GigE/iWARP and RoCE networking technologies. This software is being used by more than 2,150 organizations world-wide in 72 countries and is powering some of the top supercomputing centers in the world, including the 7th ranked TACC Stampede, 13th ranked Tsubame and the 23rd ranked Pleiades.

MPI libraries typically use the *eager* protocol for small messages and the *rendezvous* protocol for large message communication operations. MVAPICH2 uses an RDMA-based eager protocol called RDMA-Fast-Path, along with various optimizations to improve the latency of small message point-to-point communication operations. For large messages MVAPICH2 uses zero-copy designs based on RDMA-Write or RDMA-Read operations to achieve excellent communication bandwidth. Further, MVAPICH2 offers good scalability through advanced designs such as eXtended RC (XRC), Shared-Receive Queues (SRQ) and Hybrid (UD/RC) communication modes. MVAPICH2 also provides optimized collective communication using shared memory based designs. It also employs different collective algorithms based on the message and job sizes. MVAPICH2 implements one of most optimized implementations of MPI one-sided communication available on InfiniBand. Many implementations of one-sided communication are built on top of two-sided primitives. MVAPICH2 directly takes advantage of RDMA to implement Put and Get operations there by enabling applications to achieve overlap using one-sided semantics. However direct RDMA based implementation is limited to active synchronization at the time this report is written [32, 69].
2.2.2 PGAS Models

Partitioned Global Address Space (PGAS) models provide a shared memory-like abstraction that makes it easier to program applications with irregular communication patterns. They also ensure performance by exposing key locality information to the application developer. This has made them an attractive alternative to traditional message passing models like MPI. OpenSHMEM and UPC are two popular PGAS models.

2.2.2.1 OpenSHMEM

SHMEM has been a popular PGAS model. There have been several successful implementations of SHMEM including Cray SHMEM, SGI SHMEM, Quadrics SHMEM and others [72]. However, each of these implementations have their own variation of semantics making it hard for SHMEM programs to be portable. OpenSHMEM [63] is an effort to bring together these different implementations under an open standard and make SHMEM more widely useful for the community. The OpenSHMEM programming model operates on a symmetric memory address space. It allows processes or processing elements (PE) to see each other’s variables with a common name, each PE having its own local copy of the variables. These are called symmetric variables and are allocated collectively by all the PEs at the same point of the execution path. In C, symmetric objects can be global or static variables and can also be allocated dynamically from a symmetric heap using routines like shmalloc and shmemalign. OpenSHMEM defines point-to-point (contiguous and strided put/get) and collective communication operations for data movement between symmetric variables. The put operations in OpenSHMEM return when the data has been copied out of the source buffer; they need not be complete at the target. The completion at target is ensured using explicit point-to-point and collective synchronization routines. The get
operations return only when data is available for use in the local buffer and hence do not require additional synchronization. OpenSHMEM also provides atomics and lock routines that allow implementation of critical regions.

### 2.2.2.2 Unified Parallel C (UPC)

UPC is one of the most popular PGAS languages based on parallel extensions to the C language. The Berkeley UPC implementation [79] is one of the open-source implementation of UPC, which is widely ported and used in many HPC systems. A UPC application consists of UPC threads that can read, write and modify shared data in the partitioned global address. Shared data may reside in distributed memory. The underlying UPC runtime can solve and access remote memory. Remote memory accesses are often the limiting factor in parallel scalability. Therefore, the design of UPC runtime is critical to the overall performance of UPC applications. IBM and Cray also distribute their own versions of UPC implementations specifically optimized for their platforms.

The Berkeley UPC implementation utilizes GASNet interface [22] for memory access across network. It consists of Core APIs and Extended APIs. The core API interface is a narrow interface based on the Active Message paradigm. Extended APIs provide a rich, expressive and flexible interface that provides medium and high-level operations on remote memory and collective operations. GASNet supports different network conduit, including ibv (OpenIB/OpenFabrics IB Verbs), udp (UDP), lapi (IBM LAPI) [33], mpi (MPI), etc.

### 2.3 Graph500 Benchmark

The Graph500 Benchmark [78] was proposed to direct design of a new set of benchmarks that can evaluate the scalability of supercomputing clusters in the context of data-intensive applications. Graph500 benchmark stresses hardware and runtime systems by
forcing massive amounts of communication and synchronization thereby modeling more realistic application workloads. The ranking of systems based on Graph500 is released twice every year, in June and November. It consists of three comprehensive benchmarks to address application kernels: Concurrent Search, Single Source Shortest Path and Maximal Independent Set. These represent business area data sets, such as cyber security, medical informatics, data enrichment, social networks, symbolic networks, etc. We focus on the Concurrent Search benchmark in this study, which is fundamentally a Breadth First Search (BFS) traversal of the graph. This benchmark has highly irregular characteristics. The Concurrent Search benchmark consists of three phases (termed, sub-kernels, in the benchmark specification). The ‘Graph Construction’ sub-kernel constructs graph in Compressed Sparse Row (CSR) format. The second sub-kernel is the actual ‘Breadth-First-Search’. In this kernel, 64 search keys are randomly sampled from the vertices in the graph. For each of these search keys, BFS traversals are made, one by one. The final sub-kernel validates the BFS traversal. The Graph500 problem size is represented using Scale and Edge Factor. Scale is logarithm base two of the number of vertices; and, edge-factor is the ratio of the graph’s edge count to its vertex count. Thus Scale = N and and Edge factor = M indicates a graph with $2^N$ vertices and $2^N * M$ edges.

### 2.4 Out-of-core Sort Application

Out-of-core Sort application sorts the input data read from a global file system, which is vast enough such that it cannot be stored on the collective main memory of participating nodes, and outputs it into a global file system. The overview of end-to-end sort process is depicted in Figure 2.1. The active processes are divided into two distinct work groups
Figure 2.1: Overview of end-to-end Sort process

--- the read group and the sort group. The read group is a set of processes dedicated to reading input data from the global parallel file system and delivering this data in a ‘streaming’ fashion to the processes belonging to the sort group. Since the data sets are larger than the combined main memories of both groups, the input $M$ records are chunked into $q$ chunks. In read stage, the read group processes sequentially read the data from the global filesystem and transfers this data to the sort group processes. The sort group processes determine the splits based on the sampling of initial few chunks and buckets (writing the records for each split into local disk) the incoming records into these bins, stored on the local filesystem on each node. Once all the $q$ chunks have been read in, each node will have $q$ buckets stored on the local filesystem. Finally, the bucketed data is sorted and is written back to the global filesystem. During the write stage, the flow of information is mostly reversed, with the sort group processes reading the $q$ local files, one at a time, synchronized across all processes, sorting them globally, and then writing the final sorted data back to the global filesystem, as multiple files [74, 75].
2.5 Big Data Middleware

2.5.1 Memcached

Memcached was proposed by Fitzpatrick [27] to cache database request results. Memcached was primarily designed to improve performance of the web site LiveJournal. Due to its generic nature and open-source distribution [7], it was quickly adopted in a wide variety of environments. Using Memcached, spare memory in data-center servers can be aggregated to speed up lookups of frequently accessed information, like database queries, results of API calls or web-page rendering elements. Memcached clients can store and retrieve items from servers using “keys”. Identification of destination server is done at the client side using a hash function on the key. Therefore, the architecture is inherently scalable as there is no ‘central server’ to consult while trying to locate values from keys. Even though the underlying architecture of Memcached is scalable, the implementation needs careful attention in order to scale to thousands of clients accessing data simultaneously.

Figure 2.2: Memcached Architecture and Network Model
### 2.5.2 HBase

HBase is an open-source database project based on Hadoop framework for hosting very large tables [14]. It is written in Java and provides Google’s BigTable [19] like capabilities. HBase consists of three major components as shown in Figure 2.3: HBaseMaster, HRegionServer and HBaseClient. HBaseMaster keeps track of active HRegionServers and takes care of assigning data regions to HRegionServer. It also performs administrative tasks such as resizing of regions, replication of data among different HRegionServers, etc. HBaseClients check with HBaseMaster to identify which server it should request for read/write operations. HRegionServers serve client requests by fetching or updating data stored in Hadoop Distributed File System (HDFS). HDFS [1] is a fault tolerant distributed file system. Files are divided into small blocks before they are stored in HDFS. The default size of each block is 64MB, and each block is replicated at multiple DataNodes. HBase calls DFSClient to load regions from HDFS to memory. In other words, HBase acts as a client of HDFS. Usually, HBase and HDFS are deployed in the same cluster to improve the data locality.

Figure 2.3: HBase Architecture and Network Model
2.6 Big Data Middleware Benchmarks

2.6.1 Apache Olio

Olio [15] is a Web 2.0 toolkit developed at Sun Microsystems for evaluating the performance of web technologies. It defines a Social Event Calendar application and provides implementations on PHP, Java (EE) and RubyOnRails (ROR). It can be used to evaluate web technologies such as AJAX, Memcached, MobileFS, etc. It simulates a high read-to-write ratio for database accesses, which is common to social web sites. Faban [26] is a benchmark harness and driver development framework. It can generate workloads for benchmarks like Olio. Faban driver framework defines operations, transactions and associated statistics collection and reporting. It has several configuration options for selecting different web technology services, number of concurrent users, database load, etc.

2.6.2 Yahoo Cloud Serving Benchmark (YCSB)

Yahoo! Cloud Serving Benchmark (YCSB) [21] is a framework for evaluating and comparing the performance of different "Key/Value" and "cloud" serving stores. It defines a core set of benchmarks for four widely used systems: HBase, Cassandra [13], and a simple shared MySQL implementation. It also provides the flexibility for adding benchmarks for other data store implementations. There are six core workloads and each of these represents different application behaviors. For example, the typical application example for workload C is user profile cache, Zipfian and Uniform distribution modes are used in YCSB for record selection in database. Besides that, we can also define our own workloads. In addition to these different workloads, there are three runtime parameters defined in YCSB to adjust the workload on the client side, like number of clients, target number of operations per second, status report modes, etc.
2.7 Communication on Clusters with Intel MIC based Coprocessors

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

**Compute Modes:** It offers the following modes of operation for the MPI programming model [11]: 1) Offload mode, 2) Coprocessor-only mode, and 3) Symmetric mode. In **Offload** mode, MPI processes are executed on one of the architectures — either the coprocessors or the host processors. The other architecture is used solely as an accelerator for MPI processes to offload computation onto. This is akin to the usage seen with most GPGPU clusters. In offload mode, all the MPI processes execute either on the host or on the MIC with the other being used as an accelerator. In the **Symmetric** mode of operation, MPI processes uniformly span the domains of both the host and the coprocessor architectures and hence are amenable to immediate porting of existing MPI applications with little or no modifications. However, applications have to deal with the asymmetry in compute power and memory available on the MIC compared to the host. In the symmetric mode, the MPI job needs to be run in the Multiple Program Multiple Data (MPMD) mode. This is because all binaries which run on the MIC must be cross-compiled on the host for compatibility with MIC architecture. The **coprocessor-only** mode is a subset of symmetric mode with all MPI processes being confined to the MIC architecture alone.
**Communication Channels:** To support the full spectrum of usage models for MIC, Intel’s Manycore Platform Software (MPSS) offers three modes of inter-process communication: shared-memory channel, Symmetric Communication Interface (SCIF) and IB-verbs based communication. For communicating processes that reside on the same MIC device, POSIX shared memory is supported, along with a multi-threaded memcpy. Alternatively, for communication operations within the same MIC, or between the MIC and host processes, within the same compute node, SCIF generic communication API may be used. SCIF is a sockets-like API for communication between processes on MIC and host within the same system. SCIF API provides both send-receive semantics, as well as Remote Memory Access (RMA) semantics. Send-receive semantics involve both the processes, which serve as either source or destination, in the communication operation. RMA semantics define operations to register a region of memory in the user space as a window exposed
for remote access. Upon registration, further data transfers can take place in a one-sided manner with just one process either reading from a window or writing into one [11].

MPSS provides two ways of using IB verbs for communication on MIC clusters, as depicted in Figure 2.4. This allows applications to natively use MPI implementations that are built on top of InfiniBand verbs API. A direct OFED communication stack is provided to support a symmetric mode of communication on just the MIC or between the MIC and the host processor. This harnesses the advantages of the physical InfiniBand Host Channel Adapter (HCA) for intra-node and inter-node communication between a MIC and the host or between two MICs. To use IB directly and to enable processes on the MIC to talk with the HCA, Intel has facilitated a proxy based approach where all privileged operations are staged through an IB proxy client, the coprocessor Communication Link (CCL) driver, residing on the MIC to make requests on behalf of the process to IB proxy server running on the host. On completion of these privileged operations ensuing data placement calls from the process on the MIC can be made in a direct manner to the HCA using PCIe peer-to-peer copies. Alternatively, MPSS’s implementation of IB verbs over SCIF API called IB-SCIF may be used. This allows processes to use verbs API over a virtual HCA as all underlying operations are handled using SCIF. This is especially beneficial for MPI processes that reside in the Xeon Phi alone and provides ease of porting existing MPI applications to the MIC in coprocessor only mode.
Chapter 3: MVAPICH2-X: Unified Communication Runtime for MPI and PGAS

We first present our idea of unified communication runtime for MPI and UPC, the design requirements for supporting MPI and UPC, and performance evaluation results. Then we discuss the UCR enhancements and design, for supporting MPI+OpenSHMEM and present performance evaluation results.

3.1 UCR: Design Requirements and Solutions for UPC Runtime

In this section, we discuss the design requirements and solutions for the implementation of our Unified Communication Runtime (UCR) for InfiniBand clusters, for supporting MPI and UPC programming models.

3.1.1 Design Requirements

The major design requirement of our work is to enable simultaneous MPI and UPC communications without imposing any performance penalties on either MPI or UPC. That is, using our runtime, UPC programs should get identical or better performance than currently available software (and the same for MPI applications). Another design requirement is to eliminate buffer waste inside the communication library. For example, we envision
that the solution should not require separate communication resources, such as connections, buffers etc. for supporting simultaneous communications.

3.1.2 Approach

Currently, there exist several approaches that can be deployed on commodity InfiniBand clusters. We have outlined some of these approaches in Figure 3.1. Each implementation option is referred to by the name given below the stack.

The left-most alternative demonstrates that pure-MPI operations can work through the MPI standard implementations of MVAPICH on InfiniBand. The second alternative, marked “GASNet-IBV” indicates the currently available GASNet implementation on InfiniBand verbs. This implementation is on the lowest software layer made available by
InfiniBand vendors. Currently, the two approaches “MPI” and “GASNet-IBV” can be combined together to support simultaneous communications of hybrid MPI and UPC applications. However, this approach suffers from two main drawbacks: (a) Communication progress of both UPC and MPI are separate, and it is possible to deadlock UPC and MPI by not progressing in their respective communications [3] and (b) It wastes communication resources, since both MPI and UPC allocate their own connection and buffer resources.

Another approach is to use the “GASNet-MPI” stack for both MPI and UPC communications. This mode has the added advantage of progressing both MPI and UPC communications simultaneously. The primary disadvantage of this mode is that there is a mismatch of Active Messages (a fundamental design point of GASNet), and the MPI-1 point-to-point primitives. Due to this mismatch, there is a performance penalty imposed on most operations.

In our design, presented in the rightmost stack, “GASNet-UCR”, we extend the existing MVAPICH2 runtime to support native active messages. In addition, we design a new communication conduit for GASNet that supports this interface. Using this approach, both UPC and MPI communication resources are shared. Further, we have the benefit of utilizing several common communication related optimizations that have been designed for MPI over the past several years for UPC communications as well.

3.1.3 Internal Implementation Details

Our design requirements are to enable the highest possible performance for both GASNet and MPI communications. We made the following changes to implement MVAPICH2-X unified communication runtime.
Packet Headers: MPI communications stacks are required to carry message-matching information triplet (\{src, tag, context\}), to enable the tag-matching semantics of MPI. This represents an overhead of 12 bytes per packet. Additionally, packets are required to carry information for credits (such as the last received packet) for flow-control. Overall, the minimum MPI packet header is 20 bytes. We have decided that this overhead is too heavy for GASNet. Thus, we have added extra packet types in the MVAPICH2-X implementation. Accordingly, the packets used by GASNet messages are handled completely separately from the MPI packets. This approach allows us to share common optimizations from the MPI communications layer, while providing less overhead for GASNet communications. Our GASNet-UCR has only four bytes of extra header on top of GASNet-specific fields. We believe that in the future this can be improved upon as well.

Extended Interfaces: As indicated earlier in this Section, we have extended the communication implementation of the MVAPICH2-X runtime to support native Active Messages. In addition, we have also implemented the extended GASNet interfaces for natively supporting RDMA. Below is a short description of the interface.

Active Messaging Interfaces: These are the new active messaging functions that are implemented inside MVAPICH2-X.

1) int ucr_send_short_am_no_args(uint32_t dest, uint32_t token, uint16_t handler)

This function implements sending very short messages without any arguments to a remote destination. This is an optimization over the short message functions which need to
send arguments. Since the handler at the remote end expects no arguments, the space for arguments in the message packet can be optimized out.

2) int ucr_send_short_am_with_args(uint32_t dest, uint32_t token, uint16_t handler, va_list argpdr, uint8_t numargs)

   This function implements sending short messages to remote destinations. Short active messages do not carry any data payload.

3) int ucr_send_medium_msg(uint16_t dest, uint16_t token, uint8_t handler, void * source_addr, uint32_t nbytes, va_list argpdr, uint8_t numargs)

   This function implements sending medium size messages with data payload. The data payload can be sent over a bounce buffer implementation that is implemented using native RDMA [49].

4) int ucr_send_long_msg(uint32_t dest, uint32_t token, uint16_t handler, void * source_addr, uint32_t nbytes, uint32_t lkey, va_list argpdr, uint8_t numargs, void * dest_addr, uint32_t rkey, ucr_handle_t *handle_ptr)

   This function implements sending large messages. Since large messages may take a long time to send, this interface call supports returning a handle to the communication request. Using this handle, this call may be used in a non-blocking manner. Long messages are always sent using InfiniBand RDMA, with reliable connections (RC).
Extended Remote Memory Interfaces: This interface implements the GASNet extended interface. These focus on efficient remote memory access. These functions currently do not return any error status, i.e. all errors in remote memory accesses are considered fatal.

1) void ucr_inline_put (uint32_t dest, void *rem_addr, void *local_addr, size_t nbytes, uint32_t rkey, ucr_handle_t *handle_ptr)

   This function sends a very small (< 128 bytes) data payloads directly to remote memory using RDMA. This utilizes the InfiniBand “inline” send operation, which reduces sender side overhead of DMAs. As long as send work queue elements (WQEs) are available, this function completes the put on returning.

2) void ucr_put (uint32_t dest, void *rem_addr, void *local_addr, size_t nbytes, uint32_t rkey, ucr_handle_t *handle_ptr, uint32_t lkey)

   This function is used for remote put. The payload data is put directly using RDMA write. If no send work queue elements (WQEs) are available, then this data can be buffered internally (decision made based on message size), and the call returns immediately. To test for completion of the data transfer, the handle needs to be checked upon. Since the call is implemented using RDMA put, an RC connection is used.

3) void ucr_put_bulk (uint32_t dest, void *rem_addr, void *local_addr, size_t nbytes, uint32_t rkey, ucr_handle_t *handle_ptr, uint32_t lkey)

32
This function is also used for remote put. This is for supporting the GASNet put_bulk interface, in which the local memory is guaranteed to be untouched until the put operation is completed, so that data need not be buffered internally if the work queue elements (WQE) are not available.

4) `void ucr_get (uint32_t dest, void *rem_addr, void *local_addr, size_t nbytes, uint32_t rkey, uint32_t lkey, ucr_handle_t *handle_ptr)`

This function is used for remote get operations. This maps directly to RDMA Read operation, which uses RC connection (set up in an on-demand fashion).

### 3.1.4 Experimental Results

In this section, we compare performance evaluation results of the three approaches: GASNet-IBV, GASNet-MPI and GASNet-UCR described in section 3.1.2. The existing implementation of GASNet on InfiniBand verbs is called GASNet-IBV and the implementation on MPI is called GASNet-MPI. GASNet-UCR, which is our contribution in this paper, is the GASNet implementation over extended MVAPICH2-X runtime. MVAPICH-1.1 was used as the base MPI version for this work.

GASNet consists of core APIs and extended APIs. The core API interface is a narrow interface based on the Active Message paradigm. Extended APIs provide a richly expressive and flexible interface that provides medium and high-level operations on remote memory and collective operations [22]. GASNet-MPI has only the core APIs implemented whereas GASNet-IBV and GASNet-UCR have the extended APIs for one sided put and get operations. We used Berkeley UPC version 2.10.2 for our experimentation. GASNet was
configured with `--enable-pshm`, which makes use of shared memory for intra-node communication for all network conduits.

We compared these three designs from different angles. These include microbenchmark-level performance evaluation, scalability analysis based on memory footprint (memory consumption of each process as the total number of processes increase), and performance analysis of different NAS benchmarks using these conduits. Detailed analysis of these performance evaluation is presented in the following sections.

### 3.1.4.1 Experimental Platform

We used three different clusters for our experimental evaluations. Cluster A consists of four Intel Nehalem machines equipped with ConnectX QDR HCAs. Each node has eight Intel Xeon 5500 processors organized into two sockets of four cores each clocked at 2.40 GHz with 12 GB of main memory. Cluster B consists of 32 Intel Clovertown based systems equipped with ConnectX DDR HCAs. Each node in this cluster has eight Intel Xeon processors, organized into two sockets of four cores each clocked at 2.33 GHz with 6 GB of main memory. Cluster C consists of eight AMD Barcelona hosts. Each node has four sockets each with a Quad-Core AMD Opteron 8350 2GHz Processor with 512KB L2 cache and 2 MB L3 cache per core. Each node has a Mellanox MT25418 dual-port ConnectX HCA. RedHat Enterprise Linux Server 5 was used on all machines along with OFED version 1.4.2. We used Cluster A for the microbenchmark-level experiments, Cluster B for scalability analysis and Cluster C for application level evaluations.

### 3.1.4.2 Microbenchmark Level Evaluation

We chose three representative benchmarks for the microbenchmark-level performance analysis. These include the performance evaluation of UPC calls, `upc_memput`, `upc_memget`
and \texttt{bupc\_memput\_signal}. The \texttt{upc\_memput} call writes the specified amount of data bytes to the remote side. Similarly, \texttt{upc\_memget} fetches the specified amount of data bytes from remote side. \texttt{bupc\_memput\_signal} is one of the UPC extensions proposed by Berkeley Lab. It performs the same data movement semantics as that of \texttt{upc\_memput}, and updates the specified semaphore on the remote side. The update of semaphore on the remote side signals the global completion of data movement. We used Cluster A for microbenchmark performance evaluation. In this section we present results for inter-node communication. Since we configured GASNet with \texttt{--enable-pshm}, all three approaches GASNet-INCR, GASNet-MPI, GASNet-IBV will perform exactly the same inside a node. Therefore, we do not need to discuss intra-node performance in this section.

![Figure 3.2: Memput performance](image)

In \texttt{upc\_memput} and \texttt{upc\_memget} microbenchmarks, \texttt{upc\_memput} and \texttt{upc\_memget} were called in the sender thread for message sizes varying from one byte to two megabytes. The receiver thread waits on a barrier. Time taken for the UPC call for each of these message sizes is reported. We have split the latency results into two graphs, one showing the

![Figure 3.2: Memput performance](image)
Figure 3.3: Memget performance

Figure 3.4: Berkeley UPC (signal) memput performance
results for small payload sizes (1 byte to 2K bytes) and other one for large payload sizes (4KB to 2MB). Microbenchmark results are shown in Figures 3.2 and 3.3.

The results indicate that the performance of GASNet-INCR is very similar to that of the high performance GASNet-IBV. Since GASNet-MPI does not have the GASNet extended APIs, \texttt{upc\_memget} and \texttt{upc\_memput} calls are translated into active messages. These active messages are exchanged using MPI send/recv calls. This involves not only the overhead of translating into active messages, but also the extra MPI headers that these messages have to carry. But for GASNet-IBV and GASNet-INCR, \texttt{upc\_memput} and \texttt{upc\_memget} calls are translated into one sided RDMA put/get operations. This explains the huge performance difference between GASNet-MPI and the other two GASNet implementations.

We used a ping-pong test in \texttt{bupc\_memput\_signal} microbenchmark. The sender thread calls \texttt{bupc\_memput\_signal} and then calls \texttt{bupc\_sem\_wait}. The receiver thread calls \texttt{bupc\_sem\_wait} and then calls \texttt{bupc\_memput\_signal}. The \texttt{bupc\_sem\_wait} call causes the thread to wait until the semaphore gets incremented. The \texttt{bupc\_sem\_wait} call in the receiver thread gets unblocked when the data movement initiated by the sender thread is over. Similarly, the semaphore in the sender thread gets unblocked when the data movement initiated by receiver thread is over. The test was also done for message sizes varying from one byte to two megabytes. The one-way latency numbers are reported. Here also, the latency numbers are shown in two different graphs for fine-grain analysis. The observed performance results of \texttt{bupc\_memput\_signal} benchmark is similar to that of \texttt{upc\_memget} and \texttt{upc\_memput} benchmarks. The results are shown in Figure 3.4.

The microbenchmark performance results indicate that the GASNet-INCR delivers similar performance as that of GASNet-IBV conduit, and it even outperforms GASNet-IBV.
conduit for small payload sizes in \texttt{bupc\_memput\_signal} microbenchmark. GASNet-MPI performs worse in all microbenchmarks.

### 3.1.4.3 Scalability Evaluation

In order to analyze the scalability aspect of the GASNet-UCR design, we conducted memory scalability tests. In these tests, we study how the overall memory consumption of an individual process changes as the total size of the process group increases. The UPC (and MPI) programming models are “fully connected”, i.e. at any instant any thread can send a message to any other thread, and vice-versa. Therefore, it is incumbent on the runtime to either a) establish communication channels across threads before the application attempts communication, or b) provide an out-of-band technique by which communication can be set up on-demand. Since communication channels consume memory, establishing them consumes memory from the system. Another aspect of memory consumption by communication channels, is that of the choice of low-level transport.

We used a simple UPC hello world program in this experiment. We measured the memory footprint of the process, with number of connections (processes) ranging from 16 to 256. The memory footprint analysis results are shown in Figure 3.5(a). We used Cluster B for scalability analysis.

It was observed that for a 256 process UPC hello world execution, each process consumed about 227 MB in case of GASNet-UCR conduit, 265 MB in case of GASNet-IBV conduit and 237 MB in case of GASNet-MPI conduit. It can be noticed that, with increase in number of connections, memory footprint increases almost linearly for GASNet-IBV conduit, where as memory footprint of GASNet-UCR and GASNet-MPI conduits remain almost constant. The low memory footprint of GASNet-UCR and GASNet-MPI conduits
is because it uses the MVAPICH2 runtime, which is known to scale to tens of thousands of cores.

Analyzing the gradient of the scalability performance results, it can be noted that, for GASNet-IBV, the memory footprint increases by around 128KB with every additional process, whereas the memory footprint of GASNet-UCR and GASNet-MPI conduit remains constant. If we extrapolate the memory footprint for 10,000 processes, then memory footprint of GASNet-IBV will be around 1.4 GB per process, whereas the memory footprint of GASNet-UCR process will be around 250 to 300 MB per process. This shows that our design is highly scalable.

![Graph](image)

(a) Memory consumption with increasing process group size
(b) Performance comparison of UCR with MVAPICH-1.1 for MPI NAS Benchmarks on 128 processes

Figure 3.5: Scalability Analysis and MPI Performance Evaluation

3.1.4.4 MPI Application Evaluation

In this section, we present experimental results that compare pure MPI performance between MVAPICH-1.1 and MPI over MVAPICH2-X unified communication runtime. The
results are presented in Figure 3.5(b). In this figure we see the performance of MVAPICH2-UCR (denoted as MVAPICH2-INCR) normalized to that of MVAPICH-1.1, i.e. performance of MVAPICH-1.1 is designated to be 1. Based on the experimental results, we can see that MVAPICH-INCR performs virtually identically to MVAPICH-1.1. This result is expected as the extensions to support GASNet-UCR do not interfere with the MPI functionality. With this, we can conclude that the UCR library does not degrade MPI performance.

3.1.4.5 UPC Application Level Evaluation

To evaluate our GASNet-UCR design, we analyzed its performance with the NAS benchmarks [? ] written in UPC. The version of the UPC NPB Benchmark Suite used was 2.4. This version is distributed along with Berkeley UPC version 2.10.2 and can be found under upc-tests/NPB2.4. Among the NAS benchmarks, we chose to focus on Conjugate Gradient benchmark (CG), 3-D FFT PDE benchmark (FT) and the Multi-Grid benchmark (MG), for our analysis. These benchmarks were run for different problem sizes (class B and C) with 64 and 128 processes. We chose not to present experimental data for the NAS EP benchmark as it has very little communication. We used cluster C for these experiments.

The CG benchmark has the smallest problem size (75,000 for Class B and 150,000 for Class C) among the three benchmarks and has a relatively more frequent communication pattern. The communication calls from the UPC version of the NAS benchmarks mainly consist of point-to-point calls, but many of them are due to the fact that the benchmarks do not utilize UPC collectives. Rather, they implement collectives based on point-to-point operations. The execution times for these different NAS benchmarks are shown in Figure 3.6.

The performance results observed for the CG and MG benchmarks are indicated in Figures 3.6(a) and 3.6(b). For 128 process MG and CG benchmark, GASNet-UCR performs
23% and 11% faster as compared to GASNet-IBV conduit, respectively. One of the main
difference in point-to-point performance (since the benchmark does not use collectives), is
that GASNet-UCR implements a weak flow control method that is receiver driven, as op-
posed to sender keeping track of remote credits. This results in better utilization of concur-
rency in the network. Both GASNet-IBV and GASNet-UCR outperform GASNet-MPI. As
can be seen from the microbenchmark results, this is due to poorer performance of the MPI
conduit as it is trying to simulate active messages. The Class C version of the benchmarks
generate larger volume of data, which makes the runtime more dependent on bandwidth of
the network, and the differences between the three conduits is less pronounced.

We experimented with two versions of the FT benchmark, one unmodified (i.e. from
the UPC NAS 2.4 distribution), and the second was slightly modified to reduce network
hot-spotting. We modified it to remove an artificial limitation of the benchmark. The
performance results for the FT benchmark are shown in Figures 3.6(c) and 3.6(d). Fig-
ure 3.6(c) shows the performance of the unmodified version, whereas Figure 3.6(d) shows
the performance of the modified version. We note that the modified version performs sig-
nificantly better than the unmodified version, e.g. the modified version performs 21%
better than the unmodified version for FT Class C at 128 processes using GASNet-IBV
conduit. The time spent in FT is dominated by the Alltoall phase, where distributed ma-
trices are transposed. The unmodified version performs the transpose using the pattern:

```c
for(i=0;i<NTHREADS;i++) { upc_memget(i); }
```

Obviously, this results in hot-spotting threads as all threads try to simultaneously access
memory from thread 0, then thread 1 and so on. The modified version simply changes the
access pattern to distribute it better between the threads:

```c
for(i=0;i<NTHREADS;i++)
    { upc_memget(i+MYTHREAD%NTHREADS); }
```

It is evident from the results that even in
Figure 3.6: Performance of UPC NAS Benchmarks
the modified version of FT, GASNet-UCR is able to provide equal or better performance (3% better for FT, Class C on 128 processes).

3.2 UPC Queues for Scalable Graph Traversals

Distributed graph algorithms require communication between peers. In UPC, this is expressed by threads modifying globally shared data structures. It is common for multiple threads to simultaneously exchange data with a single thread. Mutual exclusion is required in such scenarios, to ensure correctness. Schemes based on locks are commonly used to achieve this. Locks on shared-memory systems are infamous for contention and come with a considerable overhead on distributed-memory architectures, therefore, they are not scalable. Another approach to this is to replicate resources and each of the remote threads operate on one of these resources. This scheme achieves mutual exclusion between sending threads but, leads to increased memory consumption and impacts performance due to the overhead of polling replicated resources. The impact of polling becomes significant as the system scale increases.

Queues provide a good abstraction for managing producer-consumer relationships. It can be implemented efficiently in a contention-free manner using active messages, while having a small memory footprint and minimal polling overhead. We propose UPC queues for scalable graph traversals. We first discuss the design requirements for queues in UPC and explain how these can be satisfied in UPC Queues. Then we present the different queue operations followed by their design and implementation using active messages.

3.2.1 Design Requirements

We expect that a design for queues in UPC should satisfy the following requirements
1) **Programmability:** Ease of programming is an important reason for the growing popularity of PGAS languages in general and UPC in particular. Ensuring this is imperative for the acceptance of any new extensions to the UPC standard. Hence we consider this the first requirement while proposing queues in UPC. Queues can improve programmability compared to the alternatives like locks and resource replication. For example, work delegation to a peer thread which involves the sequence of lock() - memput() - unlock() calls in a lock-based design can be replaced by a single enqueue call in a queue-based application. Queues can avoid the complexity of managing and polling multiple data structures for incoming work messages as is the case of a replication-based design.

2) **Scalability:** The fact that existing mechanisms for achieving mutual exclusion in UPC do not scale well is the key motivation factor behind our work. The design of queues should be centered around this requirement. It should avoid the contention seen with locks on one hand, while minimizing memory and polling overheads on the other. Active messages in GASNet provide a good option for implementing queues. As will be presented in the later sections, queues implemented over active messages can achieve very good scalability with a small memory footprint and minimum polling overhead.

3) **Low Latency:** It is necessary that queue operations have minimal overhead as compared to the lower level interface on which they are implemented. Work delegation requests, or control messages can be considered as one of the major use cases for UPC queues. It is imperative that these requests are delivered with low latency. As will be discussed in Section 3.2.3, queue operations map closely onto active messages, have minimum overheads
and can achieve latencies close to that of active messages.

4) **Portability:** As UPC is used on a myriad of system architectures, it is important for its features to be portable without sacrificing performance. The design of queues should not be based on any single network (or conduit) or specific architecture, and should ensure portable performance. We implement queues over active messages provided by GASNet. GASNet can be configured to use any of the conduits, like IBV (for InfiniBand), UDP, SMP, MPI (GASNet implementation over MPI semantics), etc. Thus, queues inherently gain this portability. Our evaluation in Section 3.2.4 shows that queues can achieve similar benefits in performance over different conduits.

### 3.2.2 UPC Queue Operations

In this section, we present an overview of the operations that can be performed on the proposed Queues in UPC. We focus on the performance and productivity aspects of queues in UPC, rather than on the syntax of queue APIs. Queue access APIs can be made in sync with UPC language constructs using efficient compiler translation techniques. We emphasize more on the concept of queues in UPC and demonstrate how applications can be implemented efficiently using it. The current implementation of queues is done in the UPC runtime layer and it supports five basic operations. Each of these are explained in detail below.

1) **upc_queue_create:** This operation creates an instance of UPC Queue and returns a handle. All subsequent queue operation calls use this handle to identify the queue instance.
This is designed as a collective call. UPC queue supports coalescing of queue elements, which avoids the communication cost for each of the enqueue operation. This feature is optional and can be enabled or disabled using specific flags in upc_queue_create.

2) upc_queue_enqueue: Queue items can be enqueued using this function. If coalescing is enabled, queue item is buffered locally, until the local bucket for the target is full or until an explicit flush call is made. Otherwise, it is sent immediately. Queue item can be any data, and the size of the data is indicated as an input parameter to this operation.

3) upc_queue_dequeue: This function call is used to dequeue items from the queue. It can operate in two modes, blocking and non-blocking. In case of blocking mode and if the queue is empty, the function call blocks until an item is put into the queue or until a specified timeout. If queue is not empty, the call returns immediately, and the queue item gets dequeued. In case of non-blocking mode, function call returns immediately whether or not queue is empty. In-out argument indicate the size of queue item, which gets dequeued. If the queue is empty, this argument is set as zero when the function call returns.

4) upc_queue_flush: This function is needed only in coalescing mode. As indicated above, in coalescing mode, queue items are buffered until the bucket is full. upc_queue_flush can be used to flush out any such buffered queue items. This is designed as a non-collective call, for better programmability.
5) **upc_queue_destroy**: Queue can be destroyed using this function call. Any resources allocated for supporting queue operations are released in this call. This is designed as a collective call.

### 3.2.3 Design and Implementation

The proposed ‘UPC Queues’ are implemented in UPC Runtime layer. UPC Runtime layer employs GASNet interface for remote memory updates and shared memory accesses. GASNet provides active messaging interface as well as direct remote memory access interface. Implementing queues over direct remote memory accesses will again require explicit locking mechanisms which have inherent performance and scalability constraints. We considered GASNet active messages for implementing UPC Queues, because active message semantics match very well with the UPC Queue implementation requirements. They provide implicit mutual exclusion when executing at a given thread.

Active messages are similar to normal messages, but it invokes a handler function at the receiver side. The handler function is selected based on the handler id, which the active message carries along with. It also carries arguments with which the handler shall be executed. GASNet Active Message interface consists of three APIs, namely, `gasnet_AMRequestShort`, `gasnet_AMRequestMedium` and `gasnet_AMRequestLong` [22]. This classification is based on message size. `gasnet_AMRequestShort` active message carries only handler id and arguments without any data payload, whereas `gasnet_AMRequestMedium` carries payload along with the handler id and arguments. `gasnet_AMRequestLong` is designed for large payloads. It puts the payload directly in the target location at the receiver side and then, the message handler is executed. The target location needs to be known while sending the message.
We chose medium active messages to implement queue operations. Small active messages do not carry payloads; and in case of long active messages, the target location needs to be known while sending the message. These do not satisfy the requirements of enqueue and dequeue operations that have a payload and operate on a queue handle rather than specific addresses. Medium active messages address these requirements well.

The enqueue operation works as follows: Enqueue operations invoked from UPC application layer is translated into GASNet medium messages at runtime layer. Queue item is set as the payload, and the handler identifier for queue operation is set as the handler id. When an active message arrives at the receiver side, it is buffered in the UPC runtime layer. Reception of active message and the buffering is transparent to the UPC application layer. The queue item is given to the application layer only when it invokes dequeue operation.
UPC Queues provide coalescing of queue items. Coalescing avoids the communication costs for each enqueue operation. Multiple queue items destined for a remote thread are aggregated and are sent out as a single active message. In order to support coalescing, separate buckets are kept for each of the remote UPC threads. As only one bucket buffer is required for each remote thread, the memory requirement is not substantial. These buckets are created during the queue creation time, based on the coalescing size specified. During an enqueue operation, the queue item is put into the bucket designated for the destination thread. The data is sent out, when the bucket is full, or when a flush operation is called by the UPC application. If enqueue operation is invoked without coalescing (immediate enqueue), the queue item is sent out immediately.

UPC Queue enqueue/dequeue operation is explained in detail in Figure 3.8. The diagram depicts the scenario when coalescing is enabled. When UPC application invokes upc_queue_enqueue, the queue item is buffered into the respective bucket for the target. This is indicated as (1) in the figure. When the bucket is full or, when an explicit upc_queue_flush is invoked, the bucket is sent over active message to the target UPC thread (2). Active message handler at the target side enqueues this into the queue (3). When the application layer at target side invokes upc_queue_dequeue, the queue item is dequeued from the queue and is given to the application (4).

The maximum payload that an active message can carry differs for different GASNet network conduits. It is determined based on performance tuning and conduit level optimizations. If the queue item size or the coalesced size is greater than the maximum payload it can carry, it is sent out in chunks. These chunks are reassembled at the receiver side, during the active message handling.
User level active message libraries like AM++ [83] have been proposed. Queue operations can also be implemented over such active message libraries. In this approach, the translation of enqueue/dequeue operations into active messages, handling of active messages, etc. need to be handled in the UPC application layer. This stands against our design consideration of programmability. Another aspect is performance. Active message libraries implemented over MPI libraries impose software overheads from MPI stack, whereas in our approach, the active messages are implemented directly over the GASNet, which does not impose additional overheads.

In this work, we focus on the concept of queues in UPC and how this can used to implement applications with irregular access patterns, in an efficient manner. We implemented queues in UPC runtime layer to demonstrate the usability and performance improvements. Using efficient compiler translation techniques, the queue concept can be implemented in sync with UPC language constructs.

![Figure 3.8: Enqueue/Dequeue operation in UPC Queue](image)
3.2.4 Experimental Results

Here, we compare the performance of the proposed UPC queues with that queue operations implemented using existing primitives in UPC. We considered UPC queue implementation using resource replication with shared notification arrays and using UPC lock primitives. We evaluate these using representative benchmarks and then discuss how UPC applications can benefit using queue approach. We start with describing the experiment platform followed by micro-benchmark and application evaluations.

3.2.4.1 Experimental Platform

We used an Intel Westmere cluster for our experiments. This cluster consists of 144 compute nodes with Intel Xeon Dual quad-core processor nodes, operating at 2.67 GHz. Each node has 12GB of memory and is equipped with MT26428 QDR ConnectX HCAs (36 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected using 171-port Mellanox QDR switch. The operating system used is Red Hat Enterprise Linux Server release 5.4 (Tikanga), with kernel version 2.6.18-164.el5 and OpenFabrics version 1.5.1.

We used Berkeley UPC version 2.12.2 [48] for micro-benchmark and application performance evaluation. This is the latest available Berkeley UPC version. In all the experiments, we used single UPC thread per process configuration mode. We used GASNet-UCR conduit for benchmark evaluation and both GASNet-UCR and GASNet-IBV conduits for application performance evaluation. GASNet-UCR is the GASNet InfiniBand conduit proposed by OSU in [38] and GASNet-IBV is the native GASNet InfiniBand conduit, implemented over InfiniBand verbs API’s. The research in [38] states that GASNet-UCR performs identical to that of the GASNet-IBV conduit in UPC level evaluations. The MPI library used in micro-benchmark evaluation is MVAPICH Library [59].
3.2.4.2 Micro-benchmark Performance

In this section we compare the performance of enqueue-dequeue operations in the proposed UPC Queues with that of implementations using existing alternatives for synchronization. We first give an overview of the different alternatives considered. Then we provide details about the benchmark and finally present the experimental results.

A common way to implement queue operations in UPC is using shared arrays and UPC locks. Even though the implementation is simple and it matches with the shared-memory programming style, such a design will not scale because of the lock contention. In the case of distributed-memory architectures, it involves a considerable communication cost. Another way to implement queues is by keeping dedicated regions in shared array for each of the remote threads and using shared notification arrays. The enqueue operation is notified using the notification array, which is also a shared array. A UPC thread polls the notification array for checking if there are any enqueue operations made by remote threads. The part of notification array that each thread polls, is local to itself so that no network operation is involved during polling. We included this design in micro-benchmark evaluation. This design is denoted as ‘UPC (notification array)’ in the graphs.

Another way for implementing the queues in UPC is using MPI Send/Receive semantics (a hybrid UPC+MPI model). Enqueues and dequeues can be translated as `MPI_ISend` and `MPI_IRecv` calls, respectively. Even though this approach looks simple, it imposes overhead from MPI software stack. This is denoted as ‘MPI’ in the graphs.

**Benchmark:** Our benchmark aims to characterize performance and scalability behavior of the different schemes discussed above. In this benchmark, every UPC thread enqueues to thread0, and thread0 on dequeueing this element, enqueues it back to the remote
thread. This pattern is characteristic of real world data intensive applications where multiple threads can simultaneously communicate with a single thread. As the number of threads increases, the contention at thread0 will help evaluate performance of the different schemes in these scenarios. Each thread enqueues 1,000 elements. We measure the average time for a single enqueue-dequeue operation.

**Latency:** Figure 3.9 compares the average time of an enqueue-dequeue operation for different queue item sizes. We conducted this experiment for different number of UPC threads. Results indicate that UPC (queue) implementation performs better than other designs, for all the message sizes. In the experiment with 128 UPC threads, enqueue-dequeue operation using the proposed queues in UPC achieves 94% lower latency when compared to an implementation using locks, for a payload of 128 Bytes. We see 74% lower latency when compared to an implementation using replication.

**Scalability:** Figure 3.10 presents the performance of queues from a scalability perspective. It shows the performance cost of enqueue-dequeue operation for a specific queue item size.
Figure 3.10: Performance comparison of enqueue-dequeue operation in different implementations of queues, for a 128 byte queue item, on varying system sizes (128 byte), for varying number of UPC-threads. The number of UPC threads is plotted on X-axis and the time for enqueue-dequeue operation is plotted on Y-axis. Results indicate that proposed UPC queue implementation scales the best.

Micro-benchmark evaluations clearly state that UPC queues do perform much better as compared to other queue implementations. It also highlights that this design does not introduce any overhead even as the number of UPC threads increases.

3.2.4.3 Graph500 Benchmark Performance

We used Graph500 version 1.2 for our experimental evaluation. We ran the benchmark for an input graph with 16 million vertices and 256 million edges, for varying number of system sizes, 64, 128, 256, 512 and 1,024 UPC-threads. We conducted this experiment with both the high performance InfiniBand GASNet conduits for UPC, GASNet-IBV and GASNet-UCR. Results of these conduits are indicated with ‘[ibv]’ and ‘[ucr]’ postﬁxes, respectively.
Performance results of Graph500 application are presented in Figure 3.11(a). Results show tremendous performance improvement for the queues version, as compared to the base version. This is because of the fact that, in queues design, we eliminate polling cost and the cost for extra message for notification. Since Graph500 is a data intensive benchmark, these costs are quite visible. Results indicate that for a 1,024 UPC-thread run, UPC queues design obtains 30% improvement over the notification array design for both GASNet-IBV and GASNet-UCR conduits. For a 512 UPC-thread run, we observe about 44% and 48% improvement for GASNet-IBV and GASNet-UCR conduits, respectively. We observe that the BFS time increases with the number of threads beyond 256 threads. This is because of the strong scaling used in this experiment.

3.2.4.4 Unbalanced Tree Search Benchmark Performance

Figure 3.11(b) shows the performance comparison of UTS benchmark (denoted as ‘UPC (base version)’) and our version using queues. We used UTS Benchmark suite v1.1 for our evaluation. The benchmark was run with an input graph of 270 billion nodes (indicated as T1WL in the benchmark specification). We ran this experiment for different
number of UPC-threads - 64, 128, 256, 512 and 1,024. Each of these were run using the InfiniBand GASNet conduits, GASNet-IBV and GASNet-UCR. We observed performance improvement for both these conduits with the queue based design. For a 512 UPC-thread run, queue version performs better than the ‘uts_upc_enhanced’ version by around 10% for both the conduits. For 1024 UPC-thread run, the performance gain is around 14% for GASNet-UCR conduit and 12% for GASNet-IBV conduit. The main reason for performance improvement is that we avoid the lock contention cost. On top of that, every lock or unlock operation results in a network communication. So for each request, there will be two network communication, whereas queue design requires only one network communication, which is the actual enqueue operation. We did not see much performance improvement for UTS benchmark as compared to the Graph500. This is because UTS benchmark is highly optimized. The lock access pattern in UTS is such that every UPC thread first tries to acquire the neighboring thread’s lock. This reduces contention among threads. We observed effects of strong scaling beyond 128 threads, similar to those observed with the Graph500 benchmark.

3.3 Optimizing Collective Communication in UPC

Collective communication primitives offer a flexible, portable way to implement group communication operations. Owing to their advantages, collective operations are supported across both MPI and PGAS models. They are also widely used across various scientific applications [35, 42]. Most MPI stacks implement collective communication using point-to-point operations. However, with the increasing use of multi-core platforms, high-performance MPI implementations have incorporated optimizations specific to multi-core architectures [23, 43, 53]. MPI implementations also rely on kernel-assisted mechanisms to
improve the performance of collective operations [51]. Even though the hybrid MPI+UPC applications can use MPI collectives to obtain better performance, pure UPC applications will still have sub-par performance [85]. Hence, it is imperative that UPC collectives be optimized.

There have been past efforts [60] in tuning and optimizing collectives in UPC, several studies [52], [68], [85] indicate the sub-par performance compared to MPI. On the other hand, MPI collectives have been well researched and have been heavily optimized over the years, as discussed above. Hence, a transparent, light-weight mechanism to map the collective operations in UPC to their MPI counterparts holds much promise to improve the performance of collective operations. Further, this allows UPC collectives to directly leverage the entire gamut of designs that are available in high-performance MPI implementations.

3.3.1 Design and Implementation

This section presents the design details of implementing UPC collectives over MPI collectives. We start this section by describing the various collectives in UPC, followed by a brief overview of the existing collective designs in Berkeley UPC (BUPC) [48] and MVAPICH2-X UPC [58]. Finally, we present our proposed design details.

The collective relocation operations defined in UPC specification v1.3 [80] are broadcast, scatter, gather, all-gather, exchange, and permute. The specification also defines computation collective operations – reduce, prefix_reduce, and sort. We focus on upc_all_broadca$t, upc_all_scatter, upc_all_gather, upc_all_gather_all, and upc_all_exchange, which are semantically equivalent to MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Allgather, and MPI_Alltoall.
Figure 3.12 provides an overview of Berkeley UPC communication stack. B UPC uses GASNet [22] as the underlying communication subsystem. In B UPC, during the UPC application compilation, remote access operations are translated as GASNet runtime operations. GASNet supports different network conduits such as MPI, native InfiniBand verbs (IBV), UDP, etc. MVAPICH2-X UPC [58] is based on Berkeley UPC and is implemented as a highly optimized network conduit for InfiniBand networks. The network conduits support active messages, and some of these even support the one-sided operations. The collective operations are all implemented in the GASNet runtime layer, using the active messages and one-sided operations.

Even though there have been efforts [60] in optimizing collective communication in UPC, the performance of UPC collectives is still not good when compared to MPI collectives [52], [68], [85]. The collectives in MVAPICH2 have been highly optimized, and use a variety of techniques such as hardware assisted, kernel based, shared memory aware, hierarchical, and topology-aware designs. They are known to scale to many thousands of cores, exhibiting very good scalability characteristics. Moreover, these designs have been highly tuned and optimized for different system configurations, system scale, message size, etc. Based on the system scale and message size, different algorithms and techniques are employed to provide the best performance. We make use of these designs for UPC collectives.
The proposed architecture is depicted in Figure 3.13. As discussed earlier, MVAPICH2-X supports MPI, PGAS, and hybrid MPI+PGAS programming models. It supports active messages and one-sided operations for implementing UPC communication operations. Furthermore, since MVAPICH2-X is based on MVAPICH2, it supports all the MPI point-to-point and collective operations. We extend MVAPICH2-X to support UPC collective operations using the existing MPI collective designs.

In MVAPICH2, collectives such as broadcast, scatter, and gather operations all leverage on two-level (intra and inter-node) hierarchical algorithms that typically make use of the k-nomial algorithm for inter-node level transfers allowing these operations to finish with $O(\log_k(N))$ asymptotic complexity [44]. When available, these collectives use hardware multicast assisted pipelined schemes for inter-node transfers. For intra-node level transfers, shared memory/LiMIC [36] assisted parallel-read/write based direct transfer mechanisms.
are used. However, for gather and scatter operations, flat-direct algorithms are used in large message and large process scale scenarios to avoid high intermediate memory buffer overheads. In the proposed design, UPC broadcast, scatter and gather operations are implemented over these schemes.

Communication intensive collectives such as all-gather and all-to-all operations use $O(\log(N))$ recursive-doubling and Bruck’s algorithms for short message transfers [43]. In addition, all-gather operations use hierarchical leader-based algorithms, with shared-memory aware communication for intra-node transfers while topology-aware designs apply to both alltoall and all-gather operations [73]. upc_all.gather_all and upc_all.exchange primitives are implemented using the all-gather and all-to-all designs in MVA-PICH2.

### 3.3.2 Experimental Evaluation

In this section, we describe our experimental testbed and discuss our evaluations. We study the performance characteristics of collective operations with our proposed designs using various micro-benchmarks for UPC collectives, and finally evaluate using UPC applications.

#### 3.3.2.1 Experiment Setup

We used TACC Stampede [77] for the performance evaluations. This cluster is equipped with compute nodes with Intel Sandybridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.3, with kernel version 2.6.32-279.el6 and OpenFabrics version 1.5.4.1.
In all the experiments, we evaluate the proposed design against Berkeley UPC 2.18.0 [48] and MVAPICH2-X UPC 2.0b [58]. The proposed design, MVAPICH2-X UPC 2.0b, and Berkeley UPC are denoted as ‘MV2X-Proposed’, ‘MV2X-2.0b’, and ‘BUPC’, respectively. For all microbenchmark evaluations, we report results that are averaged across 1,000 iterations and three different runs. In the evaluations, we do not compare with the equivalent MPI collectives, as UPC collectives operations involve handling UPC pointer semantics, and hence it will be an unfair comparison. The collective operations in the microbenchmark evaluation are invoked with the flags \texttt{UPC\_IN\_ALLSYNC} and \texttt{UPC\_OUT\_ALLSYNC}, which causes an implicit barrier before and after the collective call. The intention of this work is to optimize the collective operations in UPC, using the well researched and optimized collective designs in MVAPICH2. Comparing and contrasting programming models is clearly outside the scope.

3.3.2.2 MicroBenchmark Evaluations

In this section, we compare the performance of various collective operations in UPC, across various implementations, with a varying number of processes. The microbenchmarks used for this evaluation are designed and developed during this study; and, are available to the community along with the release of popular OSU Microbenchmark Suite [64]. In these benchmarks, the collective operations are repeated for a predefined number of times, and the average, min, and max latencies are calculated. In the following evaluations, average latency results are reported. We evaluate the performance of \texttt{upc\_all\_broadcast}, \texttt{upc\_all\_scatter}, \texttt{upc\_all\_gather}, \texttt{upc\_all\_gather\_all}, and \texttt{upc\_all\_exchange}. We compare the proposed design (MV2X-Proposed) with that of Berkeley UPC (BUPC) and MVAPICH2-X UPC (MV2X-2.0b), for varying system sizes from 128 to 2,048 UPC processes.
Figure 3.14 presents the performance evaluation of \texttt{upc\_all\_broadcast}. The results indicate similar performance for MV2X-2.0b and BUPC. This is because, both of these make use of the collective designs in GASNet, as indicated in Section 3.3.1. On the other hand, the proposed approach achieves equal or better performance, for all the message sizes. The performance of small message size is similar for the proposed design and BUPC, because both the designs employ shared memory aware collective designs. Note that, as we scale up, the improvement in performance increases. At 2,048 processes, the latency for a 256 byte message is 83.62, 81.63, and 68.17 $\mu s$ for BUPC, MV2X-2.0b, and MV2X-Proposed, respectively. The proposed design achieves significant improvement for large message sizes. For instance, at 2,048 processes, the latency reported for a 128 KB message is 7297.69, 7479.52, 299.32, $\mu s$ respectively. This is nearly 25X improvement, as compared to BUPC. The main reason for this improvement is that the proposed design collectives uses a combination of hardware based multicast, and shared memory aware hierarchical designs, as discussed in Section 3.3.1.

In Figure 3.15, we present the evaluation results of \texttt{upc\_all\_scatter}. The results are similar to the broadcast results. The proposed design performs equal or better than other designs for smaller message sizes. For larger message sizes, there is a significant improvement in performance. At 2,048 processes, the latency reported for the 512 byte scatter operation are 388.91, 362.07, and 260.68 $\mu s$, for BUPC, MV2X-2.0b and MV2X-Proposed. For 128 KB message size, the latencies are 107.08, 102.22, and 50.47 ms, respectively. This is nearly 2X improvement over BUPC and MV2X-2.0b.

The evaluation results for \texttt{upc\_all\_gather} and \texttt{upc\_all\_gather\_all} are presented in Figures 3.16 and 3.17. The gather operation copies a block of shared memory that has affinity to the $i$th thread to the $i$th block of a shared memory area that has affinity
Figure 3.14: Performance of `upc_all_broadcast`
Figure 3.15: Performance of \texttt{upc\_all\_scatter}
to a single thread. This operation is semantically similar to \texttt{MPI.Gather}. The all-gather operation copies a block of memory from one shared memory area with affinity to the $i$th thread to the $i$th block of a shared memory area on each thread. This operation is similar to the \texttt{MPI.Allgather} operation. The performance evaluation indicates that the proposed design for \texttt{upc.all.gather} operation delivers equal or better performance for small messages, and provides magnitudes of improvement for large message sizes. At 2,048 processes, the latency reported for a 256 byte gather operation is 286.13, 286.36, and 256.06 $\mu$s, respectively for BUPC, MV2X-2.0b and MV2X-Proposed. For 128 KB message size, the latencies reported are 104.35, 98.98, and 44.87 ms, respectively. The proposed design for \texttt{upc.all.gather.all} operation improves the latency of small and large messages. At 2,048 processes, the latency reported for 256 byte all-gather operation is 5678.94, 4693.66, and 903.91, respectively. At the same scale, the latency reported for 128 KB message size is 2936.74, 2570.55, and 158.36 ms, respectively. This is nearly 18X improvement in latency, as compared to BUPC.

Figure 3.18 presents the evaluation results of \texttt{upc.all.exchange}. Semantically, this operation is similar to the \texttt{MPI.Alltoall} operation. The operation copies the $i$th block of memory from a shared memory area that has affinity to thread $j$ to the $j$th block of a shared memory area that has affinity to thread $i$. The evaluation results indicate performance improvement for the proposed design, for both small and large messages at all scales. At 2,048 processes, the latency reported for a 256 byte message are 8172.99, 7538.58, and 7369.31 $\mu$s respectively, for BUPC, MV2X-2.0b and MV2X-Proposed. For 128 KB message size, the latencies observed are 3246.14, 2727.23, and 2100 ms, respectively. These are about 35% and 23% over BUPC and MV2X-2.0b, respectively.
Figure 3.16: Performance of \texttt{upc\_all\_gather}
Figure 3.17: Performance of upc_all_gather_all
Figure 3.18: Performance of \texttt{upc\_all\_exchange}
3.3.2.3 UPC Application Evaluation

We consider two application kernels — 2D-Heat and UPC NAS FT [71] — for performance evaluations. The 2D-Heat benchmark uses Jacobi method for modeling two-dimensional heat conduction. The Jacobi kernel is repeated until the standard deviation between adjacent 2D matrices is less than a predefined convergence value. In the Jacobi kernel, data transfer between adjacent processes are performed using \texttt{upc_memput} and synchronization between stages are achieved using \texttt{upc_barrier} calls. After every iteration of Jacobi kernel, sum of squares is calculated using \texttt{upc_all_reduce} calls. Process0 calculates the square root and is broadcasted to all other processes using \texttt{upc_all_broadcast}. Thus, these collective calls are invoked multiple times until the desired convergence is reached. Figure 3.19(a) depicts the performance results of 2D-Heat Transfer Modeling benchmark. We used an input matrix of size 8 K x 8 K for this experiment. The benchmark was run for varying system sizes, as shown in the figure.

As it can be noted from Figure 3.19(a), both MV2X-2.0b and MV2X-Proposed designs perform significantly better compared to BUPC. At 2,048 processes the total kernel execution time reported are 24.13, 23.74, and 10.59 seconds, respectively for BUPC, MV2X-2.0b and MV2X-Proposed. The proposed design achieves nearly 2 X improvement compared to BUPC. These results indicate same characteristics as we saw in the micro-benchmark evaluations.

The second application kernel in our performance evaluation is UPC NAS FT benchmark. The benchmark consists of an ‘all-to-all’ communication phase, and the existing version of the benchmark uses point-to-point \texttt{upc_memget} calls for implementing this phase. We modified this benchmark to use \texttt{upc_all_exchange}. The evaluation results are presented in Figure 3.19(b). We used Class C problem size for the evaluation. In UPC
NAS benchmark, classes A, B, C denote problem sizes of 128, 256 and 512, and the benchmark requires that the number of processes be less than the problem size. Thus, class C benchmark can be scaled to a maximum of 256 processes. The performance results indicate lower execution time for the proposed design, compared to BUPC and MV2X-2.0b. At 256 processes, the execution time reported are 2.99, 3.46, and 2.62 seconds respectively for BUPC, MV2X-2.0b and the proposed design. This is about 12% improvement over BUPC design.

![Graph](a) 2D Heat Benchmark (8192x8192) ![Graph](b) UPC NAS FT Benchmark

**Figure 3.19: Performance of UPC Applications**

For both the applications, the new design of UPC collectives based on MPI collectives performs better for all system sizes. The performance benefits improve as we increase the number of processes.
3.4 UCR Enhancements for OpenSHMEM Runtime

The major communication operations defined in OpenSHMEM Specification v.1.0 are data-transfer routines (One-Sided Put/Get), synchronization routines (Fence, Quiet, Barrier), collective operations (Broadcast, Collection, Reduction) and Atomic memory operations (Swap, Conditional Swap, Add and Increment). In this Section, we describe our efforts in enhancing UCR to support various OpenSHMEM communication routines. We discuss our proposed design of OpenSHMEM over UCR. In addition, we also discuss the performance and scalability benefits that can be achieved through our proposed OpenSHMEM-UCR stack for hybrid (OpenSHMEM + MPI) applications.

The earlier version of UCR [38] provided interfaces for Active Messages and One-Sided data transfer operations. We enhance UCR to support OpenSHMEM communication routines. We have added support for atomic memory operations in UCR to support compare-swap and fetch-add operations by directly leveraging InfiniBand’s native RDMA-based remote atomic memory operations. We have also enhanced UCR’s registration-cache to support remote memory registration operations.

InfiniBand requires source and destination memory addresses to be registered with Network Interface Controller (NIC). In the earlier version of UCR [38], only local memory registration keys were cached in registration cache. We have enhanced UCR’s registration-cache to support remote memory registration operations, which are required for efficiently supporting OpenSHMEM’s communication operations. The registration keys corresponding to the remote memory locations are cached in the enhanced registration cache design. During data transfer routine, symmetric address in OpenSHMEM application layer is translated to the actual address at remote process and is passed down to the communication
stack. An active message is sent to the remote process to indicate the registration require-
ment along with the memory address and size. The active message handler at remote side
registers this region and replies to the requesting process with the registration key. During
this operation, the keys get cached in both remote and local processes. Further remote mem-
ory registration requests directly get the key from the local cache, avoiding the need to send
memory registration requests via active messages, over the network. The de-registration is
done when the entry gets evicted from the cache. All these operations are transparent to the
application layer.

3.4.1 OpenSHMEM Design over UCR

In our proposed OpenSHMEM-UCR design, all the OpenSHMEM communication op-
erations are implemented over UCR. This is explained in the following Sections.

Data-Transfer Routines: OpenSHMEM data-transfer routines are implemented using
UCR’s one-sided operations. In these routines, the source/destination address of data trans-
fer operations can either be in symmetric heap or symmetric static memory, as defined in
the OpenSHMEM specification. For InfiniBand networks, the source/destination memory
must be registered with the Network Interface Controller (NIC). Since memory registration
is expensive, we use the registration-cache mechanism in UCR to amortize these overheads.

Synchronization Routines: UCR supports synchronization methods for barrier and fence
operations. These methods can be used directly for implementing OpenSHMEM barrier
and fence operations. OpenSHMEM’s Quiet routine makes sure that all outstanding remote
writes issued by a Processing Element (PE) are completed. UCR increments an internal
counter for each outstanding write request and it is decremented when the operation is
completed. Quiet is implemented by waiting until this counter is zero.
Atomic Operations: UCR provides interfaces for atomic compare-swap and fetch-add operations. All OpenSHMEM atomic memory update operations are implemented using these methods. Current OpenSHMEM implementations over GASNet employ active messages for implementing atomic memory updates. This leads to higher latency for the Fetch-add, Fetch-increment and Compare-Swap operations, because they are bounded by the time required for active messages to complete a round-trip over the network. Further, the remote host is also involved in handling these active message requests. Whereas in UCR, atomic memory update operations are implemented using InfiniBand’s RDMA-based atomic routines and the NIC updates the remote memory without requiring any involvement of the remote host processor.

Collective Operations: The collective operations defined in OpenSHMEM specification v1.0 are - shmem_barrier, shmem_collect, shmem_broadcast and shmem_reduce operations. The equivalent of these operations in MPI are MPI_Barrier, MPI_AllGather and MPI_AllReduce. Thus, we can state that OpenSHMEM collective operations are a subset of those defined in MPI. However, we cannot directly map OpenSHMEM collective operations over MPI collectives, because of the following challenges.

Difference in Specifying Participating Processes MPI uses a high-level communicator object to identify the group of processes that are participating in a collective communication operation. An MPI implementation may define special pre-defined communicators during MPI_Init and they remain valid for the duration of the parallel job. One such pre-defined communicator is the MPI_COMM_WORLD that includes all the MPI processes in the parallel job. During the course of the parallel job, new communicators can be constructed to correspond to sub-groups of processes and to define the scope of communication operations. MPI libraries create sub-communicators for each communicator object created by
the application and utilize them to improve the performance of collective operations. On
the other hand, in OpenSHMEM, there is no notion of a communicator. Each collective call
in OpenSHMEM specifies start_PE, stride_PE, and size parameters. An OpenSH-
MEM implementation uses these parameters to dynamically define the scope of a collective
operation and identify the set of processes that are participating in the specific collective
operation. Hence, an OpenSHMEM implementation is required to dynamically create log-
ical communication structures to implement collective operations, depending on the set of
parameters defined by the application. Owing to these factors, state-of-the-art OpenSH-
MEM implementations do not utilize advanced multi-core-aware designs to optimize the
performance of collective operations.

Expensive Communicator Creation A simple way to map OpenSHMEM’s collective
operations to MPI collectives is to create a new MPI communicator for each OpenSHMEM
collective operation. If an application is utilizing a unified communication library, such as,
MVAPICH2-X, it can directly invoke the corresponding MPI collective operation with such
a communicator. We note that such an operation can be performed transparently within the
OpenSHMEM implementation, requiring no modifications to an existing OpenSHMEM
application. While such a simplistic design may allow an OpenSHMEM implementation
to directly utilize MPI-level designs, it may not always deliver the best communication
latency. This is primarily because communicator creation in MPI is an expensive opera-
tion. These routines typically involve a collective communication operation between all
the participating processes to generate context-ids. Further, the OpenSHMEM implemen-
tation must carefully release the resources allocated for such communicators, in order to
prevent the MPI library from running out of internal resources. Hence, it is critical to design a light-weight interface to allow OpenSHMEM implementations to seamlessly utilize the entire range of high performance designs that are available in MPI implementations.

We take on these challenges and come up with a design with light weight *Communicator Creator* and a *Communicator Cache*. For every OpenSHMEM collective call, we propose to first check if it defines a process group to include all the processes in the parallel job. In this case, the OpenSHMEM collective operation can be directly mapped to an equivalent MPI collective routine, with the `MPI_COMM_WORLD` communicator. If the process group defined by the OpenSHMEM application does not correspond to `MPI_COMM_WORLD`, we propose to maintain a cache of communicators, *Communicator Cache*. Our design performs a look-up operation to identify a matching communicator. If our communication runtime has already created such a communicator, we consider this as a “cache-hit” and we directly re-use this communicator. However, if the process group does not correspond to any cached communicator, we treat this as a “cache-miss” and we create a new communicator using *Communicator Creator* and we cache the newly created communicator in *Communicator Cache*. Our *Communicator Creator* component relies on the non-collective communicator constructor to create the communicator in $O(\log N)$ time (where $N$ is the number of participating processes).

### 3.4.2 Hybrid (OpenSHMEM + MPI) Programming Support

Another important feature of OpenSHMEM-UCR design is the hybrid programming model support. In the current model using OpenSHMEM over GASNet, separate MPI runtime is required for supporting all MPI calls. This approach has two main draw-backs: (a) Communication progress of both GASNet and MPI are separate, and it is possible that
the communication libraries may deadlock, if they are progressed sufficiently \cite{38} and (b) 
This also leads to inefficient utilization of communication resources, since both MPI and 
GASNet allocate their own connection and buffer resources. These can result in degrading 
application performance and scalability.

In contrast, in our design, Unified Communication Runtime presents a single runtime 
which supports both MPI and OpenSHMEM communication. Such a design avoids the 
need for ensuring progress in multiple runtimes and consumes lesser network resources. 
This also removes the limitation that hybrid applications must have OpenSHMEM and MPI 
communication separated by barrier operations. Thus overlap of hybrid communication is 
possible.

3.4.3 Experimental Evaluation

In this Section, we describe our experimental test-bed and discuss our evaluations. 
We study the performance characteristics of OpenSHMEM-GASNet and our proposed 
OpenSHMEM-UCR designs with various micro-benchmarks, pure OpenSHMEM and hy-
brid MPI+OpenSHMEM applications.

3.4.3.1 Experiment Setup

We used a 144-node Intel Westmere cluster for our experiments. Each of the compute 
nodes has eight Intel Xeon cores running at 2.67 GHz with 12 MB L3 cache. The cores 
are organized as two sockets with four cores per socket. Each node has 12GB of memory 
and is equipped with MT26428 QDR ConnectX HCAs (36 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected using 171-port Mellanox QDR switch. The 
operating system used is Red Hat Enterprise Linux Server release 5.4 (Santiago), with 
kernel version 2.6.32-71.el6 and OpenFabrics version 1.5.3.
For all the experiments, we have used OpenSHMEM version 1.0a, GASNet version 1.16.2 and MVAPICH2 version 1.8RC1 communication libraries. We configured GASNet with `--enable-segment-everything` mode. We observed that the performance when configured with `--enable-segment-fast` was significantly worse for non-heap memory regions. In GASNet, shared memory communication (pshm) is disabled in segment-everything mode. In order to be fair, intra node communication mechanisms are turned off in UCR. Thus both GASNet and UCR use loop-back for intra node communication.

In the following figures, ‘OSHMEM_GASNet’ refers to the existing implementation of OpenSHMEM over GASNet and ‘OSHMEM_UCR’ represents our proposed OpenSHMEM design over UCR. We report results that are averaged across 1,000 iterations and three different runs to eliminate experimental errors.

### 3.4.3.2 Microbenchmark Evaluation

**Block Data Transfer Routines:** In this section, we study the performance characteristic of the `shmem_putmem` and `shmem_getmem` operations defined in the OpenSHMEM specification. These operations require the corresponding objects to be allocated with symmetric memory regions, which can either be statically or dynamically allocated. We have developed a set of micro-benchmarks to study the communication performance of these operations with the existing OpenSHMEM-GASNet and our proposed OpenSHMEM-UCR design. In our benchmarks, we vary the message lengths from 1 byte to 1 MB and we report the average communication latency and bandwidth for the block transfer routines for each message length. These benchmarks are run with two processes on two different nodes.

In OpenSHMEM memory model, each PE allocates a specific amount of memory for symmetric heap objects during the initialization stage. InfiniBand communication requires
Figure 3.20: Latency and Bandwidth Comparison of `shmem_getmem`

Figure 3.21: Latency and Bandwidth Comparison of `shmem_putmem`
the source and destination memory to be registered with the Network Interface Controller (NIC). To amortize the costs of the expensive memory registration operations, we register the symmetric memory regions during initialization and de-register them during finalization. However, for symmetric data objects that are statically allocated, it is not feasible to pre-register these segments. Also, registering the memory regions for each operation can significantly affect the communication performance. So, we employ the registration-cache design in the UCR for caching the keys of the frequently used memory regions. We observe that the performance of block transfer routines operating on statically and dynamically allocated symmetric objects are identical. This is because registration cache alleviates the memory registration cost in case of statically allocated symmetric objects.

In Figures 3.20(a) and (b), we evaluate the communication performance of `shmem_getmem` operation. Similarly, in Figures 3.21(a) and (b), we compare the communication performance of `shmem_putmem` operation. For brevity, we have included only small message latency results. Bandwidth results are presented for all message sizes. We observe that our proposed OpenSHMEM-UCR version performs similar or better than the OpenSHMEM-GASNet design, due to better tuned InfiniBand parameters. In these experiments, the symmetric objects are dynamically allocated. We also observed similar trends when the symmetric objects were allocated in a static manner and when we use a pool of different symmetric objects. We exclude those figures for brevity.

**Atomic Memory Routines**: We developed micro-benchmarks to study the performance of various atomic operations defined in the OpenSHMEM specification. These benchmarks repeat the atomic operations for many iterations and report the average latency. In Figure 3.22, we compare the latency of the various atomic operations between
the OpenSHMEM-GASNet and the OpenSHMEM-UCR designs. We observe that our proposed OpenSHMEM design based on UCR significantly outperforms the OpenSHMEM-GASNet version. The latency of the `shmem_cswap` routine with the UCR version is about 41% lower than that of the GASNet version. This is because the OpenSHMEM-UCR version uses RDMA atomic operations offered by InfiniBand, whereas the GASNet version implements the atomic operations over active messages. This results in a higher latency for the OpenSHMEM-GASNet design because it is bound by the latency of the round-trip time of the active messages. We observe similar performance benefits for other atomic operations, such as, `shmem_cswap`, `shmem_swap`, `shmem_fadd` and `shmem_finc`.

The `shmem_add` and `shmem_inc` operations update the remote memory atomically, but they do not wait for the reply from the remote PEs. Since these are implemented over active messages in GASNet version, the operation latency is equivalent to one-way active message latency. But, in the OpenSHMEM-UCR version, the atomic operations are implemented over InfiniBand’s atomic remote memory update routines. This leads to
an improvement of about 16% in the latency of both the shmem_add and shmem_inc
operations.

**Collective Routines:**

In Figures 3.23(a), (b), (c) and (d), we discuss performance of collective operations de-

defined in the OpenSHMEM specification with both OpenSHMEM-GASNet and OpenSHMEM-
UCR stacks. In 3.23(a), we study the performance of the barrier operation and we ob-
serve that our proposed OpenSHMEM-UCR design offers a significantly lower latency
than OpenSHMEM-GASNet. This is because the barrier algorithm in OpenSHMEM uses
the shmem_long_inc atomic operation to implement the synchronization operation and as
discussed in Section 3.4.3.2, OpenSHMEM-UCR’s atomic operations perform much better
than OpenSHMEM-GASNet’s. In Figures 3.23(b), (c) and (d), we compare the perfor-
mance of collective operations for the two OpenSHMEM versions for different number
of processes. For these evaluations, we use the benchmarks that are available in OpenSH-
MEM release and we fix the message length at 256 bytes, for all the operations. The current
version of OpenSHMEM uses linear algorithms to implement these collective operations
and also uses a barrier operation for synchronization. Hence, for these collective opera-
tions, the small message latency is strongly affected by the latency of the barrier operation.
Since OpenSHMEM-UCR delivers better latency for the barrier operation, it directly im-
proves the latency of the other collectives too. For large messages, we observed that the
communication latency of both the OpenSHMEM versions are identical. This is because
communication time for large messages becomes the major factor here and barrier time
becomes insignificant. We do not include these graphs for brevity. It is to be noted that, we
kept the same collective algorithm in both designs. Hence, the performance gain achieved
is entirely because of efficient point-to-point communication operations in UCR.
Figure 3.23: Performance of Collective Operations
3.4.3.3 OpenSHMEM Application Evaluation

In this section, we consider real-world applications that have different communication characteristics and are designed to use the OpenSHMEM model. We use the Two-Dimensional Fast Fourier Transform (2-DFFT) kernel and Two-Dimensional Heat Transfer Modeling application to evaluate OpenSHMEM-GASNet and our proposed OpenSHMEM-UCR designs. These applications have different communication characteristics.

The 2D-FFT kernel performs two-dimensional FFT on an input matrix. In this application, 2D-FFT is performed using a series of 1-D FFT’s over the rows, followed by another set of 1-D FFT’s over the columns. Each compute node performs these FFT’s for a subset of rows/columns. The application does corner-turn (distributed transpose) for re-distributing data across all the compute nodes. This phase involves an All-to-All Personalized Exchange operation. Corner-turn is done again to re-organize the data and recover the transformed output image [84]. In 2D-FFT, corner-turn operations are implemented using `shmem_putmem` and synchronization between different steps using `shmibarrier_all`.

We used an input matrix of size 8 K x 8 K for this experiment and ran using with number of processes - 32, 64, 128, 256 and 512. In Figure 3.24(a), we compare the run-times of the 2D-FFT application with both the OpenSHMEM-UCR and OpenSHMEM-GASNet versions. We observe that this application consistently performs better with OpenSHMEM-UCR when compared to OpenSHMEM-GASNet. With 512 processes, the 2D-FFT’s execution time with OpenSHMEM-UCR is about 15% better than that of the OpenSHMEM-GASNet version.

The second application which we considered is 2D-Heat Transfer Modeling. This benchmark is available in OpenSHMEM v.1.0 release. The benchmark uses Gauss-Seidel
method for modeling 2D heat conduction. Gauss-Seidel kernel is repeated until the standard deviation between adjacent 2D matrices is less than a predefined convergence value. In the Gauss-Seidel kernel, data transfer between adjacent processes are performed using `shmem_float_put` and synchronization between stages are achieved using `shmem_barrier_all` calls. After every iteration of Gauss-Seidel kernel, sum of squares is calculated using `shmem_float_sum_to_all` calls. PE 0 calculates the square root and is broadcasted to all PE’s using `shmem_broadcast64`. Figure 3.24(b) shows the performance results of 2D-Heat Transfer Modeling benchmark. We used an input matrix of size 8 K x 8 K for this experiment. The benchmark was run for different number of processes, as shown in the figure. As it can be noted from the results, OpenSHMEM-UCR version performs better than GASNet version for all scales. Moreover, the performance benefits improve as we increase the system size. With 512 processes, the execution times were 309.5 and 168.1 seconds for GASNet and UCR versions, respectively. This is about 60% improvement, which indicates that our design delivers good communication performance and scalability.

For both the applications, OpenSHMEM-UCR version performs better for all system sizes. The performance benefits improve as we increase the number of processes. The main reason for this performance gain is because of better performance in collective routines and atomic operations.

### 3.4.3.4 Hybrid Application Evaluation

In this section, we present hybrid (OpenSHMEM + MPI) application evaluation results. We evaluated two applications - 2D Heat Transmission Modeling and Graph500.

As mentioned in Section 3.4.3.3, 2D Heat Transmission Modeling application models heat transmission by executing Gauss-Seidel kernel multiple times until the standard
deviation between adjacent models reach a predefined convergence value. In the hybrid version, we used MPI routines - `MPI_Reduce` and `MPI_Bcast` for calculating and exchanging the standard deviation value. The amount of data exchanged is exactly the same as pure OpenSHMEM version. MPI collectives have been highly optimized over the past several years and performs better than OpenSHMEM collectives. This motivated us to use MPI collectives in the hybrid 2D Heat Transmission modeling application. We executed this benchmark for different system sizes as indicated in Figure 3.25(a). The Hybrid-UCR version achieves lesser execution time for all system sizes. For 512 processes, the execution times were 167.6 and 110.4 seconds for Hybrid-UCR and Hybrid-GASNet designs, respectively. This is about 34% improvement. It is interesting to note that our design out-performs the Hybrid-GASNet version even after the same set of MPI collective routines and same MPI library is being used. This is because of GASNet-Hybrid version consists of two runtimes, one for MPI and one for OpenSHMEM. For application progress, both these runtimes have to be polled. Another drawback is increased use of network resources. This is described in the following Section.

Figure 3.24: Performance of OpenSHMEM Applications
We also considered the Graph500 benchmark for evaluating the benefits of using MPI and OpenSHMEM in a hybrid environment. The Graph500 Benchmark Specification [78] is a new set of benchmarks to evaluate scalability of supercomputing clusters in the context of data-intensive applications. We use Concurrent Search benchmark kernel of Graph500 suite for hybrid application evaluation. It is basically a BFS traversal benchmark consisting of three phases. It has three sub-kernels, graph creation, BFS traversal and validation. In kernel2, 64 search keys are randomly sampled from the vertices in the graph. For each of these search keys, BFS traversals are done, one at a time. The BFS traversal kernel proceeds in a level synchronized manner. Each process discovers the neighbor vertices of all the vertices that it owns. If the newly discovered vertex is owned by the same process, it is put into a queue which will be processed in next level. If the newly discovered vertex is owned by a remote process, it is put in a local bucket and is finally sent to the remote side when the bucket is full or at the end of the level. The main communication operation in this kernel is sending buckets of vertices, which are to be processed in next level. The MPI version of the benchmark uses non-blocking MPI_ISend and MPI_IRecv routines. The bucket size used in the benchmark is 4 Kbytes.

We modified the Concurrent Search kernel to use OpenSHMEM routines, while the rest of the kernels continue to use MPI. For sending a bucket to remote process, a shmem_int_fadd operation is issued which updates the index at remote process. This updates the index with the amount of data that the PE wants to send. Then the actual data transfer is done via shmem_putmem routine. This model achieves true one-sided communication. Fetch-and-add operation enables synchronization among multiple PE’s trying to send buckets to a single remote process.
Performance evaluation results are shown in Figure 3.25(b). Hybrid-UCR design improves the performance significantly. This is because in OpenSHMEM-GASNet, fetch-and-add operation is implemented over active messages, whereas, it uses InfiniBand’s RDMA-based remote atomic operations in OpenSHMEM-design. As indicated in Section 3.4.3.2, atomic routines in the UCR design performs about 41% better than the OpenSHMEM-GASNet implementation. Similar performance gains are observed with the hybrid version of the Graph500 benchmark. For 512 processes, the UCR-Hybrid design required 87 ms to complete, whereas the run-time of the GASNet-Hybrid version was about 136 ms. This is around 35% improvement.

3.4.3.5 Memory Footprint Evaluation of Hybrid Applications

This Section presents the memory footprint analysis results. Memory footprint refers to memory consumption of a particular process. The hybrid application using GASNet requires separate MPI and OpenSHMEM runtimes. This poses increased network resource
utilization resulting in increased memory footprint. On the other hand, UCR supports both MPI and OpenSHMEM calls in a single runtime. This motivated us to evaluate memory footprint requirements for UCR-Hybrid and GASNet Hybrid versions.

Memory footprint results of UCR-Hybrid and GASNet-Hybrid runtimes are presented in Figure 3.26. We used a simple ‘Hello World’ program with `MPI_BARRIER` and `shmem_barrier_all` calls. We executed this program over different system scales, ranging from 32 to 512. As it can be observed from the figure, the memory requirement of UCR-Hybrid version is almost two-third of that of GASNet hybrid version. The difference in footprint for the two runtimes is because in GASNet-Hybrid, two runtimes have to be initialized, consuming more system resources. Eventhough the amount of extra memory required in GASNet-Hybrid is not too big, the authors believe this will affect the system performance when multiple such processes are being run. With the advent of many-core architectures, such increased memory consumption will degrade overall performance.
3.5 Optimizing Collective Communication in OpenSHMEM

Collective communication primitives offer a flexible, portable way to implement group communication operations. Owing to their advantages, collective operations are supported across both MPI and PGAS models. They are also widely used across various scientific applications [35, 42]. Most MPI stacks implement collective communication using point-to-point operations. However, with the increasing use of multi-core platforms, high-performance MPI implementations have incorporated optimizations specific to multi-core architectures [23, 28, 43, 47, 51, 53].

MPI libraries typically construct sub-communicators and implement collective operations in a hierarchical, multi-core aware manner to deliver low communication overheads. However, the communicator object is fairly robust and affects the development of irregular applications, which may involve communication operations with varying process groups. Hence, PGAS models, such as OpenSHMEM, allow applications to specify the set of processes participating in a collective in a “loosely” defined manner, through $\text{PE}_\text{start}$, $\text{PE}_\text{size}$, and $\text{logPE}_\text{stride}$ variables. An OpenSHMEM implementation can choose to construct a communication tree to implement a collective operation, such as, broadcast, based on these parameters. Or, implementations may choose to implement the collective in a simple linear manner through series of puts and gets. Invariably, such designs are not multi-core-aware, and do not use advanced shared-memory-based or kernel-assisted mechanisms thereby performing poorly when compared to their MPI counterparts. Further, the lack in performance of collective operations in OpenSHMEM will also affect the performance of parallel applications.
We explore the challenges associated with improving the performance of collective operations in OpenSHMEM. We propose a high-performance, light-weight caching mechanism in UCR, to map the primitives in OpenSHMEM to those in MPI, thereby allowing OpenSHMEM implementations to directly leverage the advanced designs that are available in MPI libraries.

We first discuss the various collectives in OpenSHMEM, followed by a discussion on the different set of challenges associated with this approach, and finally present our proposed design details and evaluation results.

The collective operations defined in OpenSHMEM specification v1.0 are - shmem_barrier, shmem_collect, shmem_broadcast and shmem_reduce operations. The equivalent of these operations in MPI are MPI_Barrier, MPI_AllGather and MPI_AllReduce. Thus, we can state that OpenSHMEM collective operations are a subset of those defined in MPI. However, we cannot directly map OpenSHMEM collective operations over MPI collectives. The challenges are explained in following section.

3.5.1 Design Challenges

**Difference in Specifying Participating Processes:** MPI uses a high-level communicator object to identify the group of processes that are participating in a collective communication operation. An MPI implementation may define special pre-defined communicators during MPI_Init and they remain valid for the duration of the parallel job. One such pre-defined communicator is the MPI_COMM_WORLD that includes all the MPI processes in the parallel job. During the course of the parallel job, new communicators can be constructed to correspond to sub-groups of processes and to define the scope of communication operations. As discussed earlier, MPI libraries create sub-communicators for each communicator.
object created by the application and utilize them to improve the performance of collective operations. On the other hand, in OpenSHMEM, there is no notion of a communicator. Each collective call in OpenSHMEM specifies start_PE, stride_PE, and size parameters. An OpenSHMEM implementation uses these parameters to dynamically define the scope of a collective operation and identify the set of processes that are participating in the specific collective operation. Hence, an OpenSHMEM implementation is required to dynamically create logical communication structures to implement collective operations, depending on the set of parameters defined by the application. Owing to these factors, state-of-the-art OpenSHMEM implementations do not utilize advanced multi-core-aware designs to optimize the performance of collective operations.

**Expensive Communicator Creation:** A simple way to map OpenSHMEM’s collective operations to MPI collectives is to create a new MPI communicator for each OpenSHMEM collective operation. If an application is utilizing a unified communication library, such as, MVAPICH2-X, it can directly invoke the corresponding MPI collective operation with such a communicator. We note that such an operation can be performed transparently within the OpenSHMEM implementation, requiring no modifications to an existing OpenSHMEM application. While such a simplistic design may allow an OpenSHMEM implementation to directly utilize MPI-level designs, it may not always deliver the best communication latency. This is primarily because communicator creation in MPI is an expensive operation. These routines typically involve a collective communication operation between all the participating processes to generate context-ids. Further, the OpenSHMEM implementation must carefully release the resources allocated for such communicators, in order to
prevent the MPI library from running out of internal resources. Hence, it is critical to design a light-weight interface to allow OpenSHMEM implementations to seamlessly utilize the entire range of high performance designs that are available in MPI implementations.

### 3.5.2 Detailed Design

We take on these challenges and come up with a design with light weight `Communicator Creator` and a `Communicator Cache`. The overall design is depicted in Figure 3.27. For every OpenSHMEM collective call, we propose to first check if it defines a process group to include all the processes in the parallel job. In this case, the OpenSHMEM collective operation can be directly mapped to an equivalent MPI collective routine, with the `MPI_COMM_WORLD` communicator. If the process group defined by the OpenSHMEM application does not correspond to `MPI_COMM_WORLD`, we propose to maintain a cache of communicators, `Communicator Cache`. Our design performs a look-up operation to identify a matching communicator. If our communication runtime has already created such a communicator, we consider this as a “cache-hit” and we directly re-use this communicator. However, if the process group does not correspond to any cached communicator, we treat this as a “cache-miss” and we create a new communicator using `Communicator Creator` and we cache the newly created communicator in `Communicator Cache`. Our `Communicator Creator` component relies on the non-collective communicator constructor, to create the communicator in $O(\log N)$ time (where N is the number of participating processes). The design details of communicator creator and cache are explained in following sections.

**Communicator Creator:** As discussed earlier, the routines provided by MPI for communicator creation are collective over an existing parent communicator. MPI-3 has proposed a routine for creating non-collective communicator. We base our communicator
Figure 3.27: Proposed Design of OpenSHMEM Collectives

creation based on this communicator creation routine. The algorithm is collective only on processes that are members of group, and the definition of the group must be identical across all the MPI processes. The complexity of this algorithm is \( \log(N) \), where \( N \) is the number of participating processes.

**Communicator Cache:** Even though the algorithm takes \( \log(N) \) steps for communicator creation, it is not advisable to create communicator during every collective call. We implemented a Least Recently Used (LRU) based cache for caching communicator. The communicators are held in this cache. When an application invokes an OpenSHMEM collective on a process group that matches a cached-communicator, we directly re-use it. The size of cache is configurable. Depending on the utilization pattern of the cached-communicators, our design transparently destroys the least recently used communicator and releases all its resources. Finally, when the application terminates, all the cached communicators are freed.
3.5.3 Experimental Evaluation

In this Section, we describe our experimental test-bed and discuss our evaluations. We study the performance characteristics of collective operations with our proposed designs, across various micro-benchmarks, pure OpenSHMEM, and hybrid MPI+OpenSHMEM applications.

3.5.3.1 Experiment Setup

We used two clusters for performance evaluations.

**Cluster A**: This cluster consists of 144 compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12 GB RAM. Each node is equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The operating system used is Red Hat Enterprise Linux Server release 6.3 (Santiago), with kernel version 2.6.32-71.el6 and OpenFabrics version 1.5.3-3.

**Cluster B**: This cluster (TACC Stampede [77]) is equipped with compute nodes with Intel Sandybridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.3, with kernel version 2.6.32-279.el6 and OpenFabrics version 1.5.4.1. Even though this system has large number of cores, we were able to gain access to only 8,192 cores.

For all the experiments, we have used MVAPICH2-X OpenSHMEM based on OpenSHMEM version 1.0vd [63]. We used Graph500 v2.1.4 in our experiment evaluations. For all microbenchmark evaluations, we report results that are averaged across 1,000 iterations and three different runs to eliminate experimental errors. We used Cluster A for
micro-benchmark and 2D-Heat application kernel evaluation and Cluster B for Graph500 evaluations.

### 3.5.3.2 MicroBenchmark Evaluations

In this section, we compare the performance of various collective operations in OpenSHMEM, across various implementations and design choices, with a varying number of processes. Specifically, we are interested in exploring the differences between OpenSHMEM’s linear and tree-based approaches for implementing collective operations and our proposed approach that allows us to map OpenSHMEM’s collectives to those in MPI. In the following figures, we refer to our proposed designs as “OSHM-Hybrid”. We also include a comparison with the latency of the corresponding MPI collective operation.

In Figures 3.28(a), (b), and (c), we study the performance of the OpenSHMEM `shmem_collect` operation, across varying number of processes. We note that the current version of OpenSHMEM reference implementation [63] does not include tree-based algorithm for `shmem_collect`. Hence, we compare OpenSHMEM’s default Linear version (denoted as “OSHM-Linear”) with our proposed Hybrid design and MVAPICH2’s default implementation of the corresponding `MPI_Allgather` collective. We note that for all the three system sizes, our proposed designs deliver significant benefits, up to 1000X times better than OpenSHMEM’s default Linear implementation. We also note that our design offers the same communication performance as the MPI implementation. Hence, we note that our proposed mapping mechanism introduces very little overheads, while allowing OpenSHMEM collectives to seamlessly leverage the efficient designs that are available in MVAPICH2.
Similarly, in Figures 3.29 (a), (b), and (c), we study the performance characteristics of different design alternatives for the `shmem_broadcast` collective operation, with varying number of processes. For this collective, we also include a comparison with the Tree-based implementation (denoted as “OSHM-Tree”) that is available in OpenSHMEM, along with the Linear implementation, our proposed Hybrid version, and the default implementation in the MVAPICH2 library. We observe that the OpenSHMEM’s tree-based design outperforms the Linear design. This is primarily because the tree implementation can allow the entire operation to complete in a maximum of $\log(N)$ steps, with $N$ OpenSHMEM processes. However, in the Linear scheme, the root of the broadcast needs to perform $N$ steps to individually transfer the data to every other process. We note that our proposed Hybrid design performs about 10X better than the tree-based implementation of broadcast in OpenSHMEM and is comparable to the performance of the default implementation of `MPI_Bcast` in MVAPICH2. These performance benefits mainly arise from the shared-memory based, hierarchical designs that are used in the MVAPICH2 library. Since such advanced designs are not available in pure OpenSHMEM implementations, our designs lead to significant performance benefits.

Finally, in Figures 3.30(a), (b), and (c), we perform a similar comparison for OpenSHMEM’s `shmem_reduce` collective operation. We note that our proposed Hybrid design outperforms OpenSHMEM’s default implementation by a factor of 100 and does not add any additional overheads, when compared to the default MPI implementation of Allreduce.

In Figures 3.31, we compare the performance of the `shmem_broadcast` operation with a different process sub-group, instead of allowing all the OpenSHMEM processes to participate in the collective operation. This benchmark is designed to demonstrate the performance of various design choices, with slightly varying process groups. As a simple
variant, we allow only the processes with even ranks to participate in the collective operation. In this version of the benchmark, we note the performance characteristics are fairly similar to those discussed in Figures 3.29(a), (b), and (c). This observation is primarily because once the new process sub-group has been formed, the new communicator is cached within the communication library and it can be re-used for any subsequent operation with the same process group.

Figure 3.28: Collect Performance Comparison (All Processes) (a) 128 Processes, (b) 256 Processes, and (c) 512 Processes

Figure 3.29: Broadcast Performance Comparison (All Processes) (a) 128 Processes, (b) 256 Processes, and (c) 512 Processes
3.5.3.3 OpenSHMEM Application Evaluation

We consider two application kernels — 2D-Heat and Graph500 [40] — for performance evaluation. The 2D-Heat application kernel is available in OpenSHMEM v.1.0 release. This benchmark uses Gauss-Seidel method for modeling 2D heat conduction. Gauss-Seidel kernel is repeated until the standard deviation between adjacent 2D matrices is less than a predefined convergence value. In the Gauss-Seidel kernel, data transfer between adjacent processes are performed using `shmem_float_put` and synchronization between stages are achieved using `shmem_barrier_all` calls. After every iteration of Gauss-Seidel kernel, sum of squares is calculated using `shmem_float_sum_to_all` calls. PE
0 calculates the square root and is broadcasted to all PE’s using `shmembroadcast64`.

Figure 3.32(a) shows the performance results of 2D-Heat Transfer Modeling benchmark. We used an input matrix of size 8 K x 8 K for this experiment. The benchmark was run for different number of processes, as shown in the figure. As it can be noted from the results, our design based on MPI collectives performs better than both linear and tree based collective designs, for all scales. Moreover, the performance benefits improve as we increase the system size. With 512 processes, the execution times were 179.6, 136.9, and 127.0 seconds for linear, tree, and hybrid designs, respectively. This is about 7% and 29% improvement, compared to tree and linear collective designs.

![Figure 3.32: Performance of OpenSHMEM Applications](image)

The second application kernel in our performance evaluation is Graph500 benchmark. The Graph500 Benchmark Specification [78] is a new set of benchmarks to evaluate scalability of supercomputing clusters in the context of data-intensive applications. We use Concurrent Search benchmark kernel of Graph500 suite in our application evaluation. It
is basically a Breadth First Search (BFS) traversal benchmark based on level synchronized BFS traversal algorithm. In the benchmark, each participating process keeps two queues — ‘CurrQueue’ and ‘NewQueue’. In each level, vertices in CurrQueue are traversed, and the newly visited vertices are put into NewQueue. At the end of each level, the queues are swapped. When the queues are empty at all the participating processes, the algorithm terminates. We modify this benchmark to use the collective reduction operation `shmem_longlong_sum_to_all` for identifying if the queues are empty. The number of times the collective operation gets called depends on the size of the graph and problem scale size. We used an input graph of size 512 million edges and eight billion vertices. We used cluster B for Graph500 evaluation.

The performance results of Graph500 hybrid benchmark are shown in Figure 3.32(b). The new design based on MPI collectives improves the performance significantly. For 4,096 processes, the new design takes about 1.81 seconds whereas the linear and tree based designs take 10.6 and 4.2 seconds, respectively. This is about 82% and 57% improvement, respectively. For both the applications, the new design of OpenSHMEM collectives based on MPI collectives performs better for all system sizes. The performance benefits improve as we increase the number of processes.

3.6 Summary

We presented our designs for Unified Communication Runtime (UCR) for MPI and PGAS on InfiniBand clusters. Our design and evaluation reveal that we are able to achieve our primary design objectives. For UPC NAS benchmarks CG and MG (class B) at 128 processes, GASNet-UCR outperforms the GASNet-IBV runtime by 10% and 23%, respectively. Further, memory scalability analysis reveals that GASNet-UCR is highly scalable.
We proposed Queues in UPC to address contention and polling overheads in data intensive and irregular applications, and presented the design and implementation of UPC Queues. Experimental results indicate that queue version for Graph500 outperforms the naive implementation by around 44% and 30% for 512 and 1024 UPC-thread runs, respectively. Performance improvements of queue variation of Unbalanced Tree Search (UTS) benchmark over the current version are about 14% and 10% for similar scale runs, respectively.

We explored the challenges associated with improving the performance of collective operations in UPC by leveraging the use of MPI collective designs, and proposed a high-performance, light-weight, and portable mechanism to map the primitives in UPC to those in MPI, thereby allowing UPC implementations to directly leverage the advanced designs that are available in MPI libraries. Our experimental evaluations show that our designs improve the performance of the UPC broadcast and all-gather operations, by 25X and 18X respectively for 128 KB message at 2,048 processes. UPC 2D-Heat performance is improved 2X times at 2,048 processes, and NAS FT (Class C) performance is improved by 12% at 256 processes.

We enhanced UCR by adding support for remote atomic memory operations and extended registration cache for allowing remote memory registrations transparently, for supporting OpenSHMEM over UCR. OpenSHMEM micro-benchmark and application evaluations reveal that our design of layering OpenSHMEM over UCR achieves very good performance over the current OpenSHMEM implementation layered on top of GASNet. Some of the microbenchmark performance highlights are 41% lower atomic operation latency, 33% and 40% improved performance for collective routines Collection and Reduction for 512 processes and 30% better performance for Barrier operation with 512 processes. OpenSHMEM application evaluations also exhibit similar performance results. 2D Heat Modeling
application execution time with 512 PE’s is improved by 45%. Further, hybrid application (OpenSHMEM+MPI) evaluation reveals that our design achieves better performance, with reduced network resource requirements as compared to hybrid MPI and OpenSHMEM layered over GASNet. For Hybrid Graph500 application executed on 512 PE’s, the BFS traversal time was improved by 35%. Memory footprint evaluations indicate that our hybrid design requires only two-third memory resources. We proposed a high-performance, light-weight caching mechanism to map the primitives in OpenSHMEM to those in MPI, thereby allowing OpenSHMEM implementations to directly leverage the advanced designs that are available in MPI libraries. Our experimental evaluations reveal that 2D-Heat performance is improved by around 7% and 29% at 512 processes, and Graph500 performance is improved by 57% and 82% at 4,096 processes over existing linear and tree based algorithms.
Chapter 4: Hybrid MPI+OpenSHMEM Design of Graph500

We present the design of hybrid MPI+OpenSHMEM Graph500 design in this section. We first discuss the existing MPI based algorithm and its implementation, and explain the overheads in existing design. Then we discuss the design challenges, and present the detailed hybrid design and performance evaluation results.

4.1 Bottlenecks in Graph500 MPI Version

Graph500 provides four MPI based reference implementations: MPI_Simple, MPI_CSR (Replicated Compressed Sparse Row), MPI_CSC (Replicated Compressed Sparse Column), and MPI_OneSided. All of these implementations use the level synchronized BFS traversal algorithm [76].

The MPI_Simple implementation is listed in Algorithm 1. In this implementation, each MPI process maintains two queues, CurrQueue and NewQueue, and two arrays — pred and visited — to store predecessor information and to track whether or not each vertex has been visited. The vertices are evenly distributed among participating processes, and only the owner process has complete information regarding adjacency list, visited array, and predecessor array for owned nodes. Initially, the ‘root’ vertex is inserted into CurrQueue by the owner process of ‘root’. An iteration of the main while loop (lines 15 to 40) corresponds to a level in the BFS traversal. In each level, the adjacent vertices
of all the vertices in $\text{CurrQueue}$ are discovered. Newly discovered edges are coalesced and sent to the corresponding owner processes using MPI_Send. Each process periodically checks for incoming data using MPI_Test / MPI_Recv and processes it (indicated as HandleReceive). Unvisited vertices in the incoming data packet are added to $\text{NewQueue}$. After processing vertices in $\text{CurrQueue}$, every process sends an empty message to all of the others to indicate end-of-level and waits until it receives empty messages from all other processes. After this ‘implicit’ barrier, an MPI_AllReduce (sum) is performed over the size of $\text{NewQueue}$. A non-zero sum indicates that there exists at least one process that has to process vertices in the next level. In this case, $\text{NewQueue}$ and $\text{CurrQueue}$ are swapped and the loop is repeated; otherwise, the algorithm ends.

The following are the three main bottlenecks in this implementation.

**Overhead in Send/Recv communication model:** Even though non-blocking MPI_Isend and MPI_IRecv are used, a lot of CPU cycles are consumed for the actual communication. To analyze this, we profiled MPI calls in the MPI_Simple implementation for 128 processes (Scale=26). The results shown in Table 4.1 indicate that more than 50% of total BFS time is spent in the MPI_Test call.

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>Total Time (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_GetCount</td>
<td>1.5</td>
</tr>
<tr>
<td>MPI_IRecv</td>
<td>20.4</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>109.0</td>
</tr>
<tr>
<td>MPI_AllReduce</td>
<td>258.0</td>
</tr>
<tr>
<td>MPI_Test</td>
<td>1100.0</td>
</tr>
<tr>
<td><strong>Total BFS Time</strong></td>
<td><strong>2040.0</strong></td>
</tr>
</tbody>
</table>

**Implicit Linear Barrier:** The implicit barrier at the end of each level is linear in nature. At a large scale, this linear barrier can cause significant overheads.

**Lack of overlap:** Even though the MPI_Simple implementation processes received edges while MPI_Isend/ MPI_IRecv calls are in progress, the actual overlap of computation-communication is low.
Algorithm 1: MPI_Simple BFS Traversal

1: \( \text{pred}[\text{root}] \leftarrow 0, \text{all\_done} \leftarrow 0 \)
2: Enqueue(\text{CurrQueue, root})
3: {\text{Procedure: HandleReceive}}
4: if \( \text{rcv\_count} = 0 \) then
5: \( \text{all\_done} \leftarrow \text{all\_done} + 1 \)
6: \textbf{else if} received then
7: \textbf{for} each received edge \((u,v)\) \textbf{do}
8: \textbf{if} \( \text{visited}[v] = 0 \) \textbf{then}
9: \( \text{visited}[v] \leftarrow 1, \text{pred}[v] \leftarrow u \)
10: Enqueue(\text{NewQueue, v})
11: \textbf{end if}
12: \textbf{end for}
13: \textbf{end if}
14: {\text{End Procedure}}
15: \textbf{while} true \textbf{do}
16: \textbf{while} \text{CurrQueue} not empty \textbf{do}
17: \textbf{for} all vertex \( u \) in \text{CurrQueue} \textbf{do}
18: HandleReceive()
19: \( u \leftarrow \text{Dequeue(CurrQueue)} \)
20: \( \text{owner} \leftarrow \text{find\_owner()} \)
21: \textbf{if} \( \text{owner} = \text{me} \) \textbf{then}
22: \( \text{visited}[v] \leftarrow 1, \text{pred}[v] \leftarrow u \)
23: Enqueue(\text{NewQueue, v})
24: \textbf{else}
25: Send \((u,v)\) to \text{owner}
26: \textbf{end if}
27: \textbf{end for}
28: \textbf{end while}
29: Send empty messages to all others
30: \textbf{while} \( \text{all\_done} \neq N - 1 \) \textbf{do}
31: HandleReceive()
32: \textbf{end while}
33: AllReduce \text{NewQueue.length}
34: \textbf{if} \text{NewQueue} is empty in all processes \textbf{then}
35: break
36: \textbf{end if}
37: Swap(\text{CurrQueue, NewQueue})
38: \text{NewQueue} \leftarrow \text{empty}
39: \text{all\_done} \leftarrow 0
40: \textbf{end while}
The MPI_CSR and MPI_CSC versions employ slightly different algorithms. In MPI_CSR, each process holds the entire graph data as a bit array. Similarly in MPI_CSC, each process has information as to whether any vertex exists in current level queue. These versions do not communicate during the level, but do an MPI_AllGather at the end of each level [76]. The MPI_OneSided version uses MPI one-sided operations for implementing BFS. The current implementation of MPI_OneSided exhibits poor performance and is excluded from our evaluations. Since the proposed hybrid design is based on MPI_Simple, the implementation details of the other versions are not discussed in detail. However, we compare our proposed design with MPI_Simple, MPI_CSR, and MPI_CSC versions.

4.2 Design and Implementation

Redesigning an MPI benchmark into a hybrid MPI+Open-SHMEM version is not trivial. A common fallacy is that replacing MPI communication routines with one-sided routines and converting the data into global arrays will just improve the performance. However, in order to extract the best performance and scalability, a meticulous design which can attain maximum communication computation overlap and reduce communication overheads is imperative. We list the challenges in designing an efficient and scalable hybrid Graph500 benchmark, discuss how we overcome these challenges, and then present our design.

4.2.1 Design Challenges

Co-ordination between sender and receiver processes: In one-sided communication semantics, the receiver process is not involved in the data transfer. So, how does the receiver process know whether the data packet has arrived? Further, how does it make sure that the entire packet has arrived and when can it start processing the data?
Co-ordination between multiple sender processes: Given the irregular nature of Graph500, how do multiple sender processes coordinate access to the target data buffer while using one-sided operations? Keeping separate receive buffers for each sender process or using locks will limit scalability [39].

Coalescing and optimal data transfer size: Since the underlying communication operations are different for MPI send/receive and Remote Direct Memory Access (RDMA) based one-sided semantics, the optimal coalescing size needs to be determined for each.

Memory scalability: While using one-sided operations for communication, sufficient buffer space must be allocated for remote memory operations. If the buffer requirement increases linearly with scale, the application will not scale. Reusing buffer requires extra synchronization between the sender and receiver processes and might incur additional overheads. Thus, optimal receive buffer size has to be determined.

Synchronization at the end of each level: We discussed in the previous section that the linear barrier causes scalability limitations. Using an MPI_BARRIER or shmem_barrier (implemented using tree-based algorithms) can improve scalability, but it limits the computation-communication overlap during the barrier. Thus, it is critical to achieve synchronization without compromising computation - communication overlap.

Load imbalance: Because of the level synchronization algorithm, any skew in the computation among processes results in a higher synchronization time. Thus, it is imperative to analyze the load imbalance and reduce it as much as possible.

4.2.2 Detailed Design

We keep the same level-synchronized BFS algorithm in our hybrid version. Thus, the communication pattern and volume in both the MPI_Simple and the Hybrid versions are
the same. We follow a flat execution model [24] for the hybrid design i.e., MPI rank and OpenSHMEM rank of the hybrid program are kept the same.

4.2.2.1 Communication using One-sided Routines

In the Hybrid design, we use OpenSHMEM one-sided routine (shmem_put) for communication. During initialization, every process allocates a globally shared buffer (allocated using shmalloc) called receive_buffer. Any process can read/write, from/to this buffer using OpenSHMEM routines. The size of the receive buffer is calculated based on the number of local edges that the process owns. Therefore the size is not dependent on the system scale and does not impose scalability concerns. Like the MPI_Simple design, the hybrid design also employs coalescing of edges.

4.2.2.2 Co-ordination using Fetch-add Atomic Operation

We use the OpenSHMEM atomic fetch-add (shmem_fadd) operation for coordinating between sender and receiver, as well as between multiple senders. The shmem_fadd atomically updates (add) the remote data and the previous value is returned. Each process maintains a globally shared variable receive_index, for its receive_buffer. Whenever a remote process wants to write data to the receive_buffer, it first atomically fetch-adds the receive_index with the write data size. After the execution of shmem_fadd, the remote process owns the region in receive_buffer and can safely write the data. Thus, an atomic fetch-add followed by a put operation achieves synchronization between sender and receiver and also between multiple senders and receiver. The receive_index is reset at the end of each level so that the receive_buffer is reused at each level.
4.2.2.3 Buffer structure for Computation-communication Overlap

Since the receiver process is not notified when data arrives, we introduced the buffer structure shown in Figure 4.1; remote processes write into receive_buffer in this format. The head and tail markers (M) indicate the beginning and end of the buffer, and the size flag (Sz) indicates the data size. The receiver process can poll on the head marker for incoming data. If the head marker is present, it will check whether the tail marker is set at an offset indicated by data size. If both markers are set, then the receiver can start processing the data. The markers are cleared after processing so that they do not signal false data arrival in the next level.

**Level synchronization using non-blocking barrier:** We use an MPI-3 non-blocking barrier for level synchronization in our hybrid design. This allows each process to enter the barrier and still continue to receive and process edges.

4.2.2.4 Load Balancing

Processing of edge \((u, v)\) includes updating the \(\text{pred}\) array, adding into \(\text{NewQueue}\), and updating the \(\text{visited}\) array if the vertex \(v\) has not been visited yet. Data structures
— predecessor array (\textit{pred}), bitwise visited array (\textit{visited}), and \texttt{NewQueue} — are kept local to the owner process. Therefore, for remote process to share the workload, it should first get the data block for processing, and update the data structures in a mutually exclusive manner. These dependencies limit the scope for load balancing in Graph500. As a result, we restrict load balancing within a node. The \texttt{shm\_ptr} routine in OpenSHMEM allows processes within a node to access globally shared memory by direct memory loads and stores. We exploit this feature in OpenSHMEM to implement load balancing. In our hybrid design, the \textit{pred}, \textit{visited} and \texttt{NewQueue} data structures are allocated in shared memory.

When a process identifies that it has more work than a predefined threshold, it exposes a portion of its \textit{receive\_buffer}. If the process that exposed work becomes idle, it can re-acquire work from its shared region. The exposed region is identified using shared variables — \texttt{p\_share\_start} and \texttt{p\_share\_stop}. Idle processes grab a portion of the exposed \textit{receive\_buffer} in a mutually exclusive manner. We considered two design alternatives for this — using OpenSHMEM atomic routines and using shared memory mutex.

Figure 4.2 illustrates mutual exclusion using OpenSHMEM atomics. An idle process does a compare-swap, \texttt{shm\_cswap}, operation at the location indexed by \texttt{p\_share\_start}, that conditionally swaps markers from head marker (M) to M1. A successful operation indicates that the idle process has gotten the chunk, and it atomically updates (using \texttt{shm\_add}) \texttt{p\_share\_start}. Since OpenSHMEM atomic routines are implemented over RDMA atomics, this design relies on the Network Interface Controller for mutual exclusion. In the shared-memory mutex-based design, every process exposes a mutex that is allocated in a globally shared region. An idle process locks the corresponding mutex of an
(a) Overloaded process exposes work

(b) One of the idle neighbor processes picks up data packet

(c) Neighbor process puts back data for post processing

Figure 4.2: Load Balancing
overloaded process to grab a portion of its receive buffer. The update operations in processing, which must be mutually exclusive with those of owner processes, will further slow down the overloaded process. So, we define processing as checking the visited array and removing all edge information, that has already been visited. Only unvisited vertices are kept for post processing by the owner process. This is indicated as ‘Post Data’ in Figure 4.2(c). The end of post data is denoted by marker M2. After preparing the post data, a special marker M3 is set at the beginning of the data packet to indicate data is ready for post-processing. The owner process keeps track of shared work and finally processes the post-data.

4.3 Experimental Evaluation

4.3.1 Experiment Setup

We used two clusters for performance evaluations.

Cluster A: This cluster (TACC Stampede [77]) is equipped with compute nodes with Intel Sandybridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.3, with kernel version 2.6.32-279.el6 and OpenFabrics version 1.5.4.1. Even though this system has large number of cores, we were able to gain access to only 8,192 cores for running experiments.

Cluster B: This cluster consists of 144 compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12 GB RAM. Each node is equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The operating system used is Red Hat Enterprise Linux
Server release 6.3 (Santiago), with kernel version 2.6.32-71.el6 and OpenFabrics version 1.5.3-3.

We used Graph500 v2.1.4 in our experiment evaluations. We used OpenSHMEM (v1.0d) [63] over GASNet (v1.20.0) [22] and MVAPICH2-X OpenSHMEM (v1.9a2) [58] as OpenSHMEM stacks. In all our Graph500 experiments, we kept the Edge Factor as 16.

**Tuning Optimal Parameters:** We tuned the optimal coalescing size for sending edges in the MPI_Simple and hybrid designs. Edges are represented as two `int64_t` elements (16 bytes). A coalescing size of 256 indicates that the edge information is sent in 4,096 byte (256*16) packets. We measured the BFS traversal times for different coalescing sizes. The optimal coalescing size identified for both MPI_Simple and Hybrid is 1,024 (16 KB), on Clusters A and B.

MPI_CSR and MPI_CSC versions are compute-intensive and rely on OpenMP threads to exploit parallelism [76]. We tuned MPI_CSR and MPI_CSC configurations with respect to the number of processes per node and the number of OpenMP threads per process. A configuration with eight processes per node and two OpenMP threads per process performed best for ‘16-core per node’ Cluster A. Similarly, configuration with four processes per node and two OpenMP threads per process performed best on ‘8-core per node’ Cluster B. We used these optimal values for all of our experiments. The MPI_Simple and Hybrid versions are executed using one process per core configurations.
4.3.2 Performance Evaluation

Figure 4.3 presents the performance results of ‘Hybrid’ MPI+OpenSHMEM versions and compares our design with pure MPI based versions. These graphs report the execution time for BFS traversal. We present results with 1,024 and 2,048 cores in Figures 4.3(a) and 4.3(b), respectively. These experiments were run on Cluster A. It can be observed from the figure that the ‘MPI_CSC’ and ‘MPI_CSR’ versions perform better than the ‘MPI_Simple’ version. But the ‘Hybrid’ version outperforms all of the MPI versions. With 2,048 cores, the time taken by the best performing MPI version (‘MPI_CSR’) was 3.13 seconds, where as the hybrid version took just 1.67 seconds. This is about 47% reduction in execution time.

![Figure 4.3: Performance Evaluation (Scale = 29)](image_url)
The communication pattern and volume in ‘MPI_Simple’ and ‘Hybrid’ versions are identical. It is the lower overhead, associated with one-sided communication calls in OpenSHMEM and efficient communication-computation overlap in the hybrid design, that resulted in better performance. In effect, the ‘Hybrid’ design reduced the total execution time from 4.5 seconds to 1.67 seconds.

**Performance Evaluation - Separate Runtimes vs. Unified Communication Runtime:**

In this section, we compare the performance of the Hybrid design executing over separate runtimes for MPI and OpenSHMEM, versus a unified communication runtime supporting both models.

Results are presented in Figure 4.4. The former is denoted as ‘Hybrid_GASNet’ and the latter is denoted as ‘Hybrid_MV2X’. The experiment was conducted on Cluster B. We used a graph with scale = 26 for this experiment. The figure demonstrates that the Hybrid_MV2X version performs significantly better than Hybrid_GASNet. For 1,024 processes, Hybrid_GASNet took 22.8 seconds and Hybrid_MV2X took just 0.58 seconds. The difference in performance is due to the following reasons: The OpenSHMEM implementation over GASNet lacks efficient atomic routine implementation compared to MVAPICH2-X. The hybrid design relies on the atomic **fetch-add** operation for acquiring a data region at the destination process. For each data transfer, there exists a **fetch-add** operation. The second reason is the overhead caused by executing two runtimes and extra network resource usage [37].

Interestingly, if the hybrid design is evaluated using separate runtimes, then the observed performance is quite low. This could mislead researchers to draw incorrect conclusions about the potential of hybrid designs. On the other hand, a unified communication runtime enables a true comparison between programming models, rather than comparing
their runtimes. Note also, that at scale=26 as shown in Figure 4.4 MPI_Simple performs better than the MPI_CSR and MPI_CSC versions unlike what is shown in Figure 4.3 for a larger graph.

### 4.3.3 Evaluation of Load Balancing

In Section 4.2.2, we discussed two design alternatives for load balancing - using pthread_mutex and using RDMA atomics. Our evaluations indicated similar performance results for both alternatives. Further, the mutex based design blocks on the lock; however if compare-swap fails, the process can proceed immediately. Because of this, we chose the non-CPU intensive RDMA atomics based scheme in our design. To measure the impact of load balancing in Graph500, we inserted an MPI_BARRIER at the end of each level and measured the time for barrier at each process. This barrier time represents the load imbalance.

We executed this modified benchmark on Cluster A and measured the total barrier time at each process in a node using HPCToolKit [31]. Figure 4.5(a) presents these results.
Figure 4.5: Effect of Work Sharing

(a) Without Work Sharing  
(b) With Work Sharing

Figure 4.6: Per Level Work Distribution
We can see that three of the processes take higher time, while the other processes finish barrier almost immediately. This indicates that those three processes have comparatively lesser work. We enabled the work sharing and evaluated load imbalance using the same experiment. The results shown in Figure 4.5(b) indicate that the work is more balanced now. However, even with work sharing enabled, we observed very little improvement in overall performance (Figure 4.8). To investigate this further, we measured the total amount of work at each level. Figure 4.6 presents this data. The results indicate that the amount of work imbalance is less and all the participating processes are busy operating on their own data. Further, the dependency on the vertex owner for processing a vertex and the higher cost for sharing work across nodes, limits the scope for work sharing in Graph500.

4.3.4 Scalability Analysis

We present strong and weak scalability evaluation results and finally present the performance results at 8,192 cores. These experiments were executed on Cluster A. Figure 4.7(a) depicts the strong scalability results. In this experiment, we kept the constant problem size (Scale = 29) and varied the scale of the system from 1,024 cores to 8,192 cores. For each scale, we measured and reported the number of edge traversals per second (TEPS). The performance results indicate that the hybrid design exhibits very good strong scalability. For MPI versions, the performance does not increase much as the system size increases because of higher communication overheads at larger scales.

Weak scalability results are presented in Figure 4.7(b). In this experiment, we kept a constant problem size per processor core as Scale=26 per 1,024 cores. With every step increase in Scale, the problem size doubles. Thus, we doubled the system size with every step in problem size. We can observe that the hybrid version of Graph500 achieves better
weak scaling results. Results at larger scale indicate that the hybrid design imposes no overheads with increase in system size.

![Graph showing weak scaling results](image)

Figure 4.7: Scalability Results

Figure 4.8 demonstrates the BFS traversal time with 8,192 processor cores. At this scale, the traversal time for MPI_Simple, MPI_CSC and MPI_CSR versions are 8.32, 2.83, and 2.68 seconds, respectively. Traversal time for the hybrid version with and without work sharing is 1.10 and 1.12 seconds. Hybrid design achieves around 59% reduction in BFS traversal time over the best performing MPI version. As mentioned earlier, the communication volume and pattern is exactly the same in both the MPI_Simple and the hybrid designs. Thus, the hybrid design is able to reduce the overall execution time from 8.32 to 1.10 seconds, which is about 8X improvement in performance.
Figure 4.8: Performance Evaluation with 8,192 Processes

4.4 Summary

We presented a detailed analysis of existing MPI based Graph500 implementation and exposed critical bottlenecks and presented a scalable and high performance design using MPI and OpenSHMEM constructs. Performance evaluations using MVAPICH2-X Unified Runtime show a reduction in Graph500 traversal time by 59%, compared to the best performing MPI design at 8,192 cores. Scalability analysis indicates that the hybrid design demonstrates good strong and weak scaling characteristics. At this scale, hybrid design performs 8X better than the MPI design which has the same communication pattern and volume. Our evaluations with a unified runtime and with separate runtimes highlight the need for a unified communication runtime for hybrid programming models.
Chapter 5: Hybrid MPI+OpenSHMEM Design of Out-of-Core Sort

We present the design of hybrid MPI+OpenSHMEM Out-of-core Sort in this section. We first discuss the existing architecture and the MPI based implementation, and the overheads in existing design. Then we present the detailed hybrid design and performance evaluation results.

5.1 Existing MPI Based Design and its Limitations

In this section, we describe the existing MPI based design and identify the major bottlenecks in it. We first describe the overall algorithm, and then describe the details.

5.1.1 Overview of Existing MPI-based Design

Figure 5.1 presents the system architecture for the MPI existing based design. The parallel job is launched such that each node has multiple processes (equal to the number of CPU cores) per node. The read group and sort group processes form two MPI communicators, READ_COMM and SORT_COMM, respectively. Because of the difference in workloads and system characteristics, only one reader process per reader host is placed in READ_COMM. In sort hosts, one process is dedicated for receiving the data from reader hosts. These receiver processes along with the sender processes in reader hosts form a third communicator — XFER_COMM. In addition to the three primary communicators defined so
To ensure that matching receives can be posted in advance, the receiving tasks in receiving tasks in read-tasks and transfer-tasks on the IO hosts. The reader hosts use OpenMP threading to cache data streamed from disk into a fifo queue for subsequent transfer to sort hosts using XFER_COMM.

Figure 5.1: Architecture for MPI based Design

Figure 5.1: Architecture for MPI based Design
far, there exists a family of binning communicators, which form a subset of $\text{SORT\_COMM}$. These $N\text{bin}$ communicators, defined as $\text{BIN\_COMM}$, are built using one process from each sort host, in a vertical manner. These $N\text{bin} \text{BIN\_COMM}$ groups are used to overlap the process of bucketing local data and saving to temporary storage with the receipt of new input data.

The reader process is threaded using OpenMP work-sharing constructs on the reader hosts such that one thread per host is dedicated solely to reading new input files and storing the streaming data in a FIFO queue. The companion transfer tasks on the same IO host in $\text{READ\_COMM}$ are in a constant spin loop checking for new data to become available. The rank 0 process in $\text{READ\_COMM}$ gathers the amount of data available on each IO host, and assigns destination ranks (receiver tasks in $\text{SORT\_COMM}$). This book-keeping by rank 0 is for transferring the data to sort hosts in a round-robin manner, among the receiver tasks. Once the destination ranks are assigned, IO tasks issues asynchronous $\text{MPI\_I}send$s with whatever data they have read so far (not exceeding the receive buffer size), to the destination sort nodes. The data size is indicated in a separate message, sent using $\text{MPI\_B}send$. The receiver tasks on sort nodes receive data in a round-robin manner, which is orchestrated by sending messages between receive tasks. That is, when a receiver task receives the message (token) from a neighboring receive task, it posts an $\text{MPI\_Recv}$ to read the length of incoming data, and then passes the token to the other neighboring task with the token updated with amount of incoming data. After passing the token, the receiver task posts an $\text{MPI\_Recv}$ to receive the actual data. The token is also used for identifying when to stop posting receives. At any point, the token indicates the total amount of data that has been received. If this equals to the total amount of expected data, no further receives are posted. The end of read stage is informed to all receiver tasks by using a special value for token.
5.1.2 Overheads/Bottlenecks in Current Design

We highlight key bottlenecks in the current MPI-based implementation of the Sort.

**Resource Utilization and Overlap:** As discussed earlier, one CPU core on each sort node is dedicated for the receiver tasks. This limits the compute resources available for the sort computation. Because of multiple BIN_COMMs, it is expected that there will be overlap between the receiver task and the computation at sort tasks. However, from profiling using HPC Toolkit [31], we see that nearly 28% of execution time of the sort task is spent waiting for the receive task, as shown in Table 5.1. This indicates non-optimal overlap and results in wasted CPU cycles on all the cores. One-sided routines using Remote Direct Memory Access (RDMA) can be employed to reduce these overheads and offload the communication to the Network Interface Controller (NIC), while freeing all the cores for computations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Percentage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation (Local Sort/Bucketing)</td>
<td>56.2</td>
</tr>
<tr>
<td>WaitForActivation</td>
<td>28.5</td>
</tr>
<tr>
<td>Final Barrier</td>
<td>15.3</td>
</tr>
</tbody>
</table>

**Book-keeping and Synchronization Overheads:** Yet another overhead is the book-keeping required at rank 0 of READ_COMM. All processes in READ_COMM continuously participate in MPI_Gather for identifying the amount of read data at each process. The rank 0 assigns destination ranks to each of these and distributes them using MPI_Scatter. The book-keeping overhead and the successive collective communication calls limit the overall read bandwidth from global filesystem.

124
Communication and Computation Load Imbalance: As indicated earlier, the sort program outputs the final sorted data in multiple files, for each bin. The output file size for each bin is a reflection of the size of data received at each bin. We analyzed the file sizes and observed a lot of variation in the file size distribution. A sample output file size distribution (with an input data of 400 GB and sorted using 256 processes) is given in Figure 5.2. The irregularity in input data received at each bin is because the sender task at each I/O host sends whatever amount of data that has been read in. Because of filesystem variations and compute power differences, this can differ, which will eventually result in unbalanced load among sort tasks.
5.2 Design and Implementation

In this section, we first present the challenges of redesigning different components of the framework for out-of-core sort using OpenSHMEM. We then present the details of the overall design.

5.2.1 Design Challenges

Read Task Coordination with Reduced Synchronization: As outlined in Section 5.1.2, in the MPI-based implementation, read tasks use collective communication to determine the amount of data read by each read task and the mapping between read tasks and sort tasks. A goal of the proposed design is to reduce overheads by removing the collective synchronization where possible. The challenge here is to establish coordination using one-sided communication operations over global shared memory.

Read Task-Write Task Synchronization and Overlap: In the proposed framework, we would like to remove the need for dedicated receive processes, thereby increasing the available computer power. We would like to divide the transfer between read tasks and sort tasks into multiple chunks so that the data movement can be overlapped with the sort process at each sort task. This requires a light-weight mechanism for the read tasks to identify the buffers to write to, while the sort tasks are busy computing.

One-sided Semantics and Detecting Remote Completion: The two-sided semantics inherently provide a notification to the receiving sort task, when the data is available. When one-sided communication is used, an additional mechanism is required for the sort tasks to identify the completion of transfers. Similarly, a mechanism is required for the read task to know when the remote buffer is free to write into. However, this should not involve remote polling as it increases network traffic and is detrimental to performance.
5.2.2 Proposed Hybrid Design Architecture

Figure 5.3 shows the modified architecture we propose for the hybrid MPI+OpenSHMEM based design of out-of-core sort. The use of global memory model and one-sided communication removes the need for dedicated receive processes, thereby adding more compute power to the sort group. We introduce multiple data staging buffers between the read group processes and the sort group processes to enable overlap between data movement and the sort computation. The coordination among the read group and selection of destination sort processes for data transfer is established using an atomic counter based design. We first present an overview of the logical global memory management and then go into the details of different components of the framework later in this section.
5.2.3 Detailed Design

5.2.3.1 Global View of Memory and Management

Each process in the PGAS model exposes specific amount of memory that any process can access (read/write). Such global view of data can be utilized for efficient data transfer mechanisms. Figure 5.4 depicts the organization of global memory allocated at all processes, for the hybrid sort application.

The memory region that each process exposes is split into multiple buffer chunks of 100 MB each. These buffers are used for data transfer. In addition, a statusbuffer region is also maintained in the shared heap region to maintain the status of each data buffer. Similar to the current MPI based design, only one process per read host acts as the active read process while the other remain passive. This choice is due to the fact that one process alone is able to saturate the limited read bandwidth available per node and a large number of simultaneous readers can degrade the overall performance of the underlying Lustre filesystem [75]. However, this requires the active read process to use all the memory available on the node. To this end, the active read process takes control of the communication buffers allocated by all the other processes in the node, using the shmem_ptr functionality available in OpenSHMEM. This enables the active read process to directly load and store data into this memory, even though it is hosted by other processes. Due to the globally visible memory at the sort processes, the read processes can also write data directly to the destination sort process rather than requiring dedicated receive processes, like in the earlier framework.
5.2.3.2 Read Group

In the existing framework, all the active read processes synchronize using an MPI_Gather operation to collect the amount of data each process has read. Then the rank 0 among read hosts assign destination ranks based on the gathered data. This assignment is dispersed to all the read hosts using an MPI_Scatter operation. These are repeated until all the input data have been transferred. In the proposed design, we avoid these book-keeping overheads by using a centralized atomic counter. When a read process has a data block ready to be transferred, it atomically increments the counter by one. The distribution of data among the sortprocs is done in a round-robin fashion in a vertical manner. Note that the bucketing of data happens in a synchronized manner among the BIN_COMM sort processes, as explained in Section 5.1. To identify the sort process destination, each read process employs logic described in Equation 5.1.

\[
dest\_{\text{rank}} = (\text{numIoHosts} \times \text{numTasksPerHost}) + ((\text{counter}/\text{numSortHosts}) \mod \text{numTasksPerHost}) + (\text{counter} \mod \text{numSortHosts}) \times \text{numTasksPerHost}
\]  

(5.1)
Where, the `numIoHosts`, `numSortHosts`, `numTasksPerHost` indicate number of reader hosts, number of sort hosts, and number of tasks per host, respectively. This approach completely removes the collective synchronization used in the earlier framework.

Similarly, the buffer index at the destination process where the data needs to be placed can be calculated using Equation 5.2.

\[
buffer\_index = \left(\frac{counter}{(numSortHosts \times numTasksPerHost)}\right) \mod \left(\frac{MAX\_READ\_BUFFERS}{numTasksPerHost}\right)
\]

(5.2)

Where, `MAX_READ_BUFFERS` indicates the number of buffers allocated per host. In our experiments we chose the number of buffers to be five, and can be configured through environmental parameters.

### 5.2.3.3 Sort Group

Using the buffer identification logic, the `sort` process can expect the data in its local buffers, in a round-robin manner. This avoids the need for polling multiple buffer locations. Further, the proposed framework removes the need for dedicated receiver processes by enabling direct data delivery to all the `sort` processes.

### 5.2.3.4 Data Delivery

The data delivery mechanismemployed in the hybrid design is explained in Figure 5.5. As discussed earlier, the `read` host which has data ready to send atomically increments the counter using OpenSHMEM atomic `fetch-add` operation. Based on the counter value, the destination `sort` process and the buffer location is identified. But, before writing
the data to the remote side, the read process needs to make sure that the remote buffer is empty, and the sort process has finished processing the buffer. As discussed earlier in Section 5.2.1, remote memory polling mechanisms shall not be used as it will degrade the performance, and increase network traffic. We use atomic compare-swap routine to ensure buffer availability.

The read process does a compare-swap operation on the remote buffer status, which compares the value with 1, and if success, swaps it with $p+\text{rank}$. Here $p$ denotes the total number of processes, and $\text{rank}$ denotes the rank of the read process. The compare-swap operation returns the original value. Thus, a return value of ‘0’ indicates that the buffer is free, and then the read process proceeds with put+notify operation. This operation puts the data into buffer, and writes the value 1 at buffer status location. On the other hand,
if the return value of `compare-swap` operation is 1, then it indicates that the particular `sort` process has not yet processed the buffer. In this case, the `read` process will wait on polling a local memory location, which is symmetric to the remote buffer status location. Finally, when the `sort` process finishes processing of buffer, it identifies that the buffer status is \( p + 1 \), and it unblocks the waiting `read` process by updating the symmetric buffer status location at `read` process (identified using rank information.) The buffer structure is explained in Figure 5.6.

5.3 Performance Evaluation

In this section, we present detailed evaluation and analysis of data transfer time, overlap of computation and communication, output file size distribution, and final sort time.
5.3.1 Experimental Setup

We used TACC Stampede [77] for our experimental evaluations. The compute nodes are equipped with Intel SandyBridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX2 HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.3, with kernel version 2.6.32-279.el6 and OpenFabrics version 1.5.4.1. The global filesystem in Stampede is Lustre v 2.1.3. Even though this system has large number of cores, we were able to gain access to only 4,096 cores for running experiments.

The algorithms and software developed to support the end-to-end disk sorting procedure were all written in C++. The MPI based sort code used in this study is described in [75]. We used MVAPICH2 v2.0a as the underlying MPI library for pure MPI-based design. We used MVAPICH2-X Unified Communication Runtime for MPI+PGAS based on v2.0a for the hybrid design. The OpenSHMEM stack in MVAPICH2-X v2.0a is based on OpenSHMEM v1.0d.

All of the performance evaluations are measured on the system running in normal, production operation via batch job submission. Since the global file systems are a shared resource, available to all users for general purpose I/O in their applications, the I/O bandwidth delivered to our sort procedure is not guaranteed to be constant. Hence, we ran our experiments multiple times, and the lowest numbers are reported for both MPI and hybrid designs.
5.3.2 Evaluation of Different Phases of Sort Operation

In this section, we evaluate the execution times of different phases in sort. As discussed in Section 5.1, there are two stages in overall sorting procedure: 1) global filesystem read and transfer, and 2) final sort and writing to global filesystem. We measure the individual execution times for each of these stages, for both MPI based design and the proposed hybrid design. These results are presented in Figure 5.7.

Here, ‘TX’ indicates the transfer stage and ‘FS’ indicates the final sort stage. The value in parenthesis indicates the input size that the experiment was run with: 500 GB and 1 TB. As it can be observed from the figure, the hybrid design reduces the execution time of data transfer stage significantly. The execution time for data transfer stage for 1 TB is 776 and 440 seconds for MPI and hybrid designs, respectively. This represents an improvement of 1.76 times over the pure MPI based design. The improvement is mainly because of the one-sided communication and reduced overhead in case of hybrid design. Further, there is an extra bin of sort tasks (which are used as dedicated receiver tasks in MPI based design), which can also work on local sorting and bucketing.

The execution time for final sort stage for 1 TB data size is 546 and 462 seconds, respectively. As indicated in Section 5.2, the hybrid design does the data transfer in a uniform manner, such that each sort task receives almost equal amount of data. This is achieved by the use of atomic counter and sending the data (using remote put) in uniform chunk size. This uniform data distribution improves the load balancing among the sort tasks. The improvement in sort stage for hybrid design is about 15%.
5.3.3 Overlap Evaluation and Resource Utilization

In this experiment, we evaluate the extent of overlap of computation and communication. Here, the computation refers to the local sorting and bucketing of the records into local file system. Overlap is computed as the total time for computation divided by the overall time. If the computation and communication are perfectly (100%) overlapping, the value will be one. We measure the individual time for local sorting, binning and bucketing and thus compute time.

We conducted this experiment for input size of 100 GB, 150 GB and 200 GB. The experiment was run using 256 processes, with 4 reader hosts and 12 sort hosts. The overlap results are presented in Figure 5.8. The MPI based design achieves about 70% overlap, and the hybrid design could further increase the overlap to nearly 100%, as the input size increases.
The hybrid design uses one sided communication operations leveraging RDMA feature. Thus, there is no need for explicit receiver tasks in sort nodes, as compared to the two-sided MPI model. These compute cores are utilized for sorting in hybrid design, thereby increasing the compute power.

### 5.3.4 Distribution of Output File Size

As discussed earlier in Section 5.1.1, the distribution of output file size has a direct co-relation with the distribution of input files sizes. Further, we observe that in MPI based design, the amount of data transferred in each send might differ, and this impacts the load balancing among the sort tasks in a bin. In this section, we compare the distribution output file sizes for the same input size (200 GB) for the hybrid and pure MPI based designs. This comparison is presented in Figure 5.9. As it can be observed from the figure, the output file size distribution is much more uniform in hybrid design. Notice that there are periodic spikes in output file sizes. These are the files generated by the last process in each bin group.
These file sizes are expected to be larger than other files, as per algorithm characteristics. However, due to better load balancing done by the hybrid implementation, the spikes are lesser for the hybrid case.

### 5.3.5 Overall Execution Time of Sort

Figure 5.10 presents the total execution time for the sort operation, for varying input sizes. These experiments are weak scaling experiments, where we keep a constant problem size of 1 TB per 1,024 CPU cores. We evaluated this experiment on varying system sizes — 512 cores to 4,096 cores.

For 4 TB input size, the execution times observed for MPI and hybrid designs are 2408 and 1172 seconds, respectively. This is about 51% reduction in execution time. Yet another interesting observation here is that the rate of increase in execution time with input size is
higher for MPI based design. On the other hand, increase in execution time for hybrid design is lesser. Thus the proposed design exhibits good weak scaling characteristics.

We note that the execution time for 2 TB input size is slightly higher. We plan to rerun this experiment and will update the results for camera-ready submission.

5.3.6 Scalability Evaluation

We evaluate the strong scaling characteristics of the proposed design in this section. Here, we keep the problem size constant (500 GB), and execute on varying system sizes — 512 cores to 4,096 CPU cores. We report the sorting rate, represented as TB per minute, and is presented in Figure 5.11.

Evaluation results indicate that the increase in sorting rate for hybrid design is much higher, as compared to the original MPI based design. This indicates that hybrid design exhibits good strong scaling characteristics. At 4,096 cores, the sort rate reported are 0.163 and 0.355 TB/min, for MPI and hybrid designs, respectively.
5.3.7 Verification of Sort Output

We verified the output of the sort operation obtained from the Hybrid design to ensure correctness. Since the hybrid design transfers data in a uniform manner, the output file sizes are different for both designs, as explained in Section 5.3.4. Thus, for ensuring correctness, we merged the output files into a single file, and compared the `md5sum`. The `md5sum` values in both designs were identical.

5.4 Summary

We identified various bottlenecks in the MPI-based implementation of k-way Sample-Sort and presented the challenges involved in redesigning the data delivery using the OpenSHMEM PGAS model. We proposed a scalable and high performance design of sort using hybrid MPI+OpenSHMEM models and proposed simple extensions to OpenSHMEM communication. To the best of our knowledge, this is the first such design of any data intensive
computing application using the Hybrid MPI+PGAS programming model. Our performance evaluations reveal that the hybrid design improves the performance significantly. At 4,096 processes, the sort execution time is reduced by 51% using the hybrid design as compared to original MPI based design. Our scalability experiments indicate that hybrid design demonstrates good strong and weak scaling characteristics.
Chapter 6: Extending OpenSHMEM for Intel MIC Clusters

Intel Many Integrated Core (MIC) architectures are becoming an integral part of modern supercomputer architectures due to their high compute density and performance per watt. Partitioned Global Address Space (PGAS) programming models such as OpenSHMEM, provide an attractive approach for developing scientific applications with irregular communication characteristics, by abstracting shared memory address space, along with one-sided communication semantics. However, the current OpenSHMEM standards do not efficiently support heterogeneous memory architectures such as MIC. Owing to the different memory and compute characteristics of MIC and Host, application developers might want to execute kernels with non-uniform memory requirements across them. But, the global symmetric memory allocation in current OpenSHMEM standard mandates same amount of memory to be allocated on every process.

In this section, we present the proposed extensions to alleviate this restriction and propose high performance runtime level designs for efficient communication involving MIC processors. We also present application-level case studies to demonstrate the effectiveness of the proposed designs and extensions. Application redesigns using the proposed extensions provide several magnitudes of performance improvement, as compared to the symmetric mode.
6.1 Extending OpenSHMEM to Support Non-Uniform Memory Allocation

As discussed in Section 1, the current OpenSHMEM standards support only equi-sized symmetric memory allocations, and mandates that all the processes participate in the call. This model results in wastage of memory resources for applications with memory load imbalance. Moreover, the existing model cannot be used for heterogeneous memory architectures. Given the fact that the emerging HPC architectures have heterogeneous memory systems, it is imperative to have an efficient solution for supporting variable-size symmetric memory allocation in OpenSHMEM. We first list the functional and non-functional requirements for such a solution, and then present the proposed extension.

We note that the proposed extension shall be applicable to generic uniform (such as host-only or native mode) and non-uniform memory architectures. Further, the variable memory allocation should be transparent to communication operations, and should have little overhead for programmer. Moreover, there should not be any difference in the way these memory regions are accessed during communication operations, and shall allow communication across all memory regions. Finally, the solution shall be applicable for upcoming memory/processor architectures.

**Proposed Extension: Team-based Memory Allocation:** We propose team-based symmetric memory allocation for handling non-uniform memory allocation requirements. We also introduce teams into OpenSHMEM. In the current OpenSHMEM standards, there is no notion of teams. However, the concept of teams have been there for collective operations for MPI. Also, some flavors of SHMEM (such as CraySHMEM use teams for collective communication operations. We propose to introduce teams into OpenSHMEM, and extend it to use for non-uniform memory allocations.
Figure 6.1 provides different team based routines. `shmem_team_create` creates a new team (`new_team`) based on the ranks and size information. This is a collective call, with all the processes within team. Similarly, `shmem_team_split` splits the team into `new_team`, based on the color and key arguments. `shmem_team_rank` and `shmem_team_size` provide the size and rank information of a particular team. `shmem_team_destroy` destroys a previously created team. Special constants such as `SHMEM_TEAM_WORLD` and `SHMEM_TEAM_SELF` indicate global and self teams, respectively, as analogous in case of MPI. For more details about team routines, please refer to [12].

```c
void shmem_team_create(shmem_team_t team, int *ranks, int size, shmem_team_t *newteam);
void shmem_team_destroy(shmem_team_t *team);

void shmem_team_split(shmem_team_t team, int color, int key, shmem_team_t *newteam);

int shmem_team_rank(shmem_team_t team);
int shmem_team_size(shmem_team_t team);
```

Figure 6.1: Proposed Application Interfaces for Team

We propose to use teams for non-uniform memory allocations, with the following application interfaces depicted in Figure 6.2. The `shmalloc_team` call will allocate `size` amount of memory on each of the processes, which is a member in `team`. A successful invocation of `shmalloc_team` call will return a valid memory address on processes which
are in the team, and a handle on processes outside the team. The non-team processes can use this handle for reading/writing the symmetric memory allocated during this call. The handle can be used exactly the same way as that of remote destination address, in the case of OpenSHMEM communication operations. The runtime will translate these handles to actual symmetric memory locations.

```c
void *shmalloc_team (shmem_team_t team, size_t size);
void shfree_team(shmem_team_t team, void *addr);
```

Figure 6.2: Team-based non-uniform memory allocation

The proposed solution can handle all the requirements stated earlier. In order to allocate different amount of memory on processes running on host and co-processor, application developer just need to create teams of host and co-processor processes, followed by shmalloc_team calls with appropriate sizes. Similarly, host-only applications with variable memory requirements can just form teams, and can have appropriate memory allocated with required size. We also note that this solution can be extended to future architectures with higher memory/processor hierarchies. We illustrate the use of team-based shmalloc with the following pseudo-code depicted in Figure 6.3.

In this example, some of the processes (listed in heavy_ranks) have to allocate a big amount of memory. For this, first a team with these processes is created using
shmem_team_create(SHMEM_TEAM_WORLD, heavy_ranks, num_heavy_ranks, &heavy_team);

void *heavy_ptr = shmalloc_team(heavy_team, big_size);
...
shmem_putmem(heavy_ptr+0x1000, src_addr, len, some_heavy_rank);

Figure 6.3: Example of shmalloc_team usage

shmem_team_create. Following the team creation, memory allocation on these processes can be made by invoking shmalloc_team call. This call returns a handle (valid address on all the processes in heavy_ranks, and special handle on all other processes), which can be used for subsequent communication operations, just like a normal pointer. Thus the putmem call writes data pointed by src to the memory allocated at some_heavy_rank, at an offset of 0x1000.

6.2 OpenSHMEM Runtime Design

In this section, we present the design and implementation of the proposed extensions, and communication runtime designs for efficiently supporting communication involving MIC processes.

6.2.1 Memory Manager for Team-based Alloc

As discussed in previous section, the proposed shmalloc_team call is collective across all the processes, and the operation returns either a valid address (on all the processes
in the team) or a handle (on processes outside the team). We define the handle such that it can identify the remote memory location. The structure of the handle is depicted in Figure 6.4. The handle has three parts – the ‘magic-bits’, ‘identifier’, and the ‘offset’. The ‘magic-bits’ is used by the runtime to differentiate between a normal address and one which points to a team-shmalloced memory region. These bits are selected such that these are outside the heap and static memory regions. Further, a unique identifier is assigned for each shmalloc_team call, and an entry is made into a book-keeping table. Since the allocation call is a collective call, every process can uniquely identify each allocation. The book-keeping table keeps track of the offsets in symmetric heap, at which memory is allocated at each process. This requires a collective all-gather communication, among all the processes. Note that, for the existing shmalloc call, the OpenSHMEM runtime does an all-gather collective operation to check that the size requested by all processes is the same. Thus, the costs of both shmalloc and shmalloc_team calls are similar. The final part of the handle – ‘offset’ – is set as ‘0’ during the shmalloc_team call. This is provided so that the handle can be used to reference a region inside the allocated region.

![Figure 6.4: Structure of Address Handle](image)

With this design, the handles can be used just like normal memory addresses in OpenSHMEM communication routines. Further, the handles can be used to denote offsets within
the symmetric memory allocations. During a communication operation, the runtime checks if it is a handle, by looking for the magic bits. If it is a handle, the allocation entry is identified using the identifier-bits. Finally, communication operation is applied to the corresponding memory address at offset identified by offset bits. During shfree_team call, the entry is deleted from the book-keeping table.

6.2.2 Communication Runtime

In this section, we discuss the proposed runtime designs for improving the performance of intra- and inter-node OpenSHMEM communication operations on Intel MIC clusters. The proposed communication runtime efficiently takes advantage of both InfiniBand (IB) and PCIe channels.

The Intel MPSS software stack allows access to the InfiniBand HCA from inside the Xeon Phi coprocessors through its CCL (Coprocessor Communication Link) driver. We refer to this as ‘DirectIB’. As Xeon Phi is connected to the host as a PCIe device, the data movement between Xeon Phi and the IB HCA is carried out as peer-to-peer communication over the PCIe bus. We note from our previous study on MPI communication on Intel MIC [66] that the current generation processor chipsets from Intel provide limited peak bandwidth for peer-to-peer communication, especially when reads from MIC memory are involved. The InfiniBand HCA directly issues memory read/write operations when data needs to be exchanged with remote MIC or remote Host. Owing to the hardware limitation, this leads to very low communication performance.

MPSS stack provides the Symmetric Communication Interface (SCIF) that provides communication over the PCIe bus, between a Xeon Phi and its Host. SCIF has been shown to deliver close to peak PCIe bandwidth performance through its RMA (Remote Memory
Access) semantics. One way to work around the limitations with DirectIB is to stage data through the host, by taking advantage of the SCIF and host-to-host IB channel. However, if the process resides on the Xeon Phi, we need an agent/proxy on the host to stage its data. It is important that the communication staged through the host makes efficient use of both the PCIe and IB channels while the proxies cause minimal intervention to any application processes running on the host. DirectIB provides a low latency path for inter-node communication from the MIC while host-based staging provides a high bandwidth path. Similarly, shared memory channel provides a low latency path for intra-node communication. It is critical for communication runtimes to provide a hybrid design that takes advantage of the various channels, based on the communication characteristics. We discuss the runtime designs for various OpenSHMEM routines in the following subsections.

6.2.2.1 Direct IB for small messages

The CCL interface in MPSS supports direct InfiniBand communication. This provides a low-latency path for small message communication. We use DirectIB for small message transfers involving MIC processes. However, the limitation in peer-to-peer bandwidth severely hinders the communication performance for large messages. These limitations are expected to be more prominent when a data transfer operation (shmem_put) is issued from a mic-process, since this involves a read by the IB adapter from the mic-process memory. In the following sections, we explore design alternatives to improve communication performance between remote MIC processes.

6.2.2.2 Proxy-based design for large message transfers from MIC

As discussed in Section 1, the transfers from MIC memory incurs bandwidth limitations with the current generation architectures. The SCIF interface in Intel MPSS offers a
high bandwidth scheme for transfers involving host and MIC processes, within the node. We take advantage of the alternative channels, for designing a high performance communication runtime, avoiding all the bandwidth bottlenecks. In our previous study [66], we had proposed proxy-based schemes for two-sided MPI communication. These schemes are not applicable directly in OpenSHMEM, because of the one-sided communication semantics. Both the sender and receiver processes participate in the proxy-based communication, that was proposed in previous study. We enhance this design to support one-sided communication. Every node running MIC processes will have one additional process, which handles the communication requests from MIC processes. We first discuss the proposed design for MIC-to-MIC communication, and then the MIC-to-Host /Host-to-MIC communication using proxy process.

![Diagram](image.png)

(a) Put Operation (MIC-to-MIC)  
(b) Get Operation (MIC-to-MIC)

Figure 6.5: Proxy based design for OpenSHMEM communication operations

When a MIC process (initiator) wants to write data to a remote (target) MIC process, the initiator MIC process sends a request to the proxy process running on local host. The proxy process reads the data from the source process over SCIF channel. This read operation is pipelined with writes to the target process memory over the InfiniBand interface. This is depicted in Figure 6.5(a). Similarly, for MIC-to-MIC read operations, the initiator
MIC process sends a request to the proxy. The proxy reads data from the peer process over SCIF, and writes to the *initiator* process memory over InfiniBand channel, in a pipelined manner (depicted in Figure 6.5(b)).

Proxy-based design is also employed for communications involving MIC and host processes. For put operations from MIC to host, the MIC process sends a request to proxy, and the proxy reads the data from MIC over InfiniBand. Note that the read bandwidth initiated from host process is not limited. This data is directly placed on to the host processes memory. During the runtime initialization, the proxy process registers the heap memory region of all the host processes running in that node. This allows zero-copy transfer (reading from remote MIC memory, and writing to destination host process) using proxy. Similarly, reads initiated from host process for reading from remote memory is also being serviced by proxy. The initiator host process sends the request to the proxy running on the MIC process. The proxy process reads from the MIC process over SCIF, and writes to the *initiator* host process directly over InfiniBand, in a pipelined manner.

The proposed proxy based design can be turned on or off at runtime. We provide this flexibility since not all applications require proxy. Users can enable or disable proxy based on application’s communication characteristics.

6.2.2.3 Intra-node Communication Designs

As discussed earlier, the MPSS stack provides the SCIF interface that provides communication over the PCIe bus, between a Xeon Phi and its Host. SCIF has been shown to deliver close to peak PCIe bandwidth performance through its RMA (Remote Memory Access) semantics. However, the latencies of small message transfers is higher than the shared memory transfers, and even DirectIB transfers. However, SCIF provides better performance for large messages compared to both shared memory and DirectIB channels.
Thus, in the proposed runtime, we carefully switch between appropriate channel (shared memory, SCIF, and DirectIB), based on the message size, and transfer direction.

### 6.3 Redesign of Graph500 using the Proposed Extensions

The Graph500 [78] benchmark represents the subclass of data intensive and irregular applications that use graph algorithm-based processing methods. In our previous work [40], the we proposed a scalable and high performance hybrid MPI+OpenSHMEM design of Graph500. We provide a brief overview of this design, and then explain the redesign of this benchmark using the proposed extensions.

The Graph500 benchmark is a level based Breadth First kernel. In the existing implementation, each process maintains two queues, `CurrQueue` and `NewQueue`. At each level, vertices in the `CurrQueue` are processed and the newly discovered vertices are sent to the owner processes. The receiving process adds the vertex to the `NewQueue` only if it is not visited already. At the end of each level, the queues are swapped, and proceeds to next level. The algorithm terminates when the length of `NewQueue` is empty at all the processes.

We propose to enhance the existing design such that the processing of vertices in `CurrQueue`, and processing of received vertices and preparation of `NewQueue` can be done in parallel. For this, we propose to use the MIC cores for processing the received vertices, and preparing the `NewQueue`. The host process sends vertices to MIC processes on the same node as that of the owner host process. Note that, with this design, we only need to allocate buffers for receiving vertices from remote host processes, and not the memory intensive graph data structures. Thus, we utilize the proposed extensions and allocate only receive buffers on MIC processes. MIC cores process the received vertices, and prepare the
NewQueue by the time the host process processes vertices in CurrQueue. Since multiple cores are available in MIC, NewQueue can be processed in parallel in a multi-threaded manner. However, using multiple threads involves additional challenges in synchronization, otherwise synchronization costs will supersede the benefits of parallelism.

The need for synchronization arises because multiple threads try to insert into NewQueue. Also, before adding into the queue, the predecessor array (global) needs to be updated. Validation errors are possible when multiple threads add into NewQueue and update predecessor array, since the predecessor information can get overwritten. Heavier synchronization operations such as mutex-locks shall be avoided for performance reasons. We propose a merger thread based design. In this design, every thread keeps a local queue. Before adding vertices into the local queue, it checks for predecessor array if the element is already added. If not, the predecessor array is updated by putting negative value of the vertex. The merger thread runs in background and merges the local queues to form the NewQueue. During merging, it checks for the corresponding entry in the predecessor array, and sets it to positive value. This way, duplicates are avoided during merging, and predecessor arrays are updated correctly. Further, this design does not need any synchronization among threads.

6.4 Experimental Results

In this section, we describe our experimental setup and evaluate the performance of various data movement, and collective operations in OpenSHMEM, and application level performance for symmetric and native modes. We use OSU OpenSHMEM micro-benchmarks (v4.3) for the point-to-point and collectives evaluation. To study the performance aspects of symmetric and native modes, we present application level performance evaluations using Graph500. Finally, we also present the results of Graph500, redesigned using the proposed
extensions to OpenSHMEM. In all the experimental evaluations, we compare the proposed design with that of the default MVAPICH2-X OpenSHMEM version 2.0rc1. We denote the default design, and the proposed designs as ‘MV2X-Def’, and ‘MV2X-Proposed’, respectively.

6.4.1 Experimental Setup

We use the Stampede supercomputing system at TACC [77] for our experiments. Each Stampede node is a dual socket containing Intel Sandy Bridge (E5-2680) dual octa-core processors, running at 2.70GHz. It has 32GB of memory, a SE10P (B0-KNC) coprocessor and a Mellanox IB FDR MT4099 HCA. The host processors are running CentOS release 6.3 (Final), with kernel version 2.6.32-279.el6.x86_64. The KNC runs MPSS 2.1.4346-16. The compiler suite used is the Intel Composer_xe_2013.2.146.

6.4.2 Performance of Point-to-Point Operations

We present the performance results of OpenSHMEM \texttt{shmemput} and \texttt{shmemget} operations, while running on MIC and host processes, for both intra- and inter-node cases. The intra-node put latency results are presented in Figure 6.6. We cover all the three cases: Host-to-MIC, MIC-to-Host, and MIC-to-MIC. We use DirectIB for put operations from host to MIC, as it offers highest bandwidth. Since MV2X-Def uses InfiniBand channel, the performance numbers for both designs are similar. However, for MIC-to-Host, and MIC-to-MIC, we can see that the IB channel offers very limited bandwidth, and the latencies are high. MV2X-Proposed efficiently uses SCIF communication channel for large messages, and offers lower latencies for both MIC-to-Host and MIC-to-MIC. The put operation latencies observed for 4 MB message for intra-node MIC-to-MIC transfer is 5013.4 and 703.9
respectively for MV2X-Def, and MV2X-Proposed designs, which is 7X reduction in latency.

Inter-node put performance results are presented in Figure 6.7. Similar to the intra-node results, we can see that DirectIB channel offers higher latencies for large message transfers from MIC-to-Host and MIC-to-MIC. As indicated in Section 1, SCIF channel is available only for intra-node transfers. Thus the proposed design relies on active-proxy for these transfers, where the data is staged to the proxy at the destination node, and then transferred to the target process over SCIF channel, in a pipelined manner. With active-proxy, the latency of large message transfer is significantly reduced. The put operation latencies observed for inter-node MIC-to-MIC transfer are 3911.8 and 838.7 $us$ respectively for MV2X-Def, and MV2X-Proposed designs, which is 4.6X improvement in performance.

![Figure 6.6: Intra-node Put Latency](a) Host to MIC (b) MIC to Host (c) MIC to MIC

The intra- and inter-node performance results of OpenSHMEM shmem_get operation are presented in Figures 6.8 and 6.9. The performance characteristics are similar to that of put. However, here we can see that get operations initiated from hosts have higher overheads in the DirectIB implementation used in MV2-Def. This trend is apparent for
intra- and inter-node cases. For intra-node, the proposed design utilizes SCIF channel, and improves the communication performance. For inter-node, the get requests from host is forwarded to the active-proxy residing on the remote process. The active-proxy then reads the data from the corresponding MIC process using SCIF channel, and writes them into the originating process directly over RDMA operations. The SCIF-reads and IB-write operations are pipelined to achieve best latency. The intra-node MIC-to-MIC latencies for get operation with 4 MB message size is 4962.0 and 741.7 us, for default and proposed designs, respectively. Thus the proposed design reduces the latency by a factor of 6.7X. The inter-node MIC-to-MIC latencies for get operation with 4 MB message size is 3889.83 and 837.2 us, for default and proposed designs, which is 4.6X improvement.

6.4.3 Performance of Collective Operations

The collective operations defined in OpenSHMEM are - shmem_barrier, shmem_collect, shmem_broadcast, and shmem_reduce operations. We present the the performance characteristics of these operations, when run on host and MIC processes. In all the evaluations, we used 16 processes running on both host and MIC per node (i.e., 32 processes per node).
Figure 6.8: Intra-node Get Latency

Figure 6.9: Inter-node Get Latency
We vary the number of nodes, and the overall latency of the operation for different message sizes is reported.

Figure 6.10 presents the broadcast operation latencies for varying system sizes. As it can be observed from the figure, the proposed design improves the latency significantly, for both large and message sizes. The collective operations are implemented over point-to-point operations, and thus the lower latencies in point-to-point operations result in better performance for collective operations. Further, the proposed design relies on host-staged designs for collective operations [45, 81]. The broadcast operation latencies observed for 4 KB message size for 1,024 processes is 130.4 and 21.8 \( \mu s \) for default and proposed designs, which is a factor of six improvement.

Similar performance characteristics are observed for other collective operations. The performance evaluation results of these operations are presented in Figures 6.11, 6.12 and 6.13. At 1,024 processes, the operation latency observed for 4 KB reduce operation is 1084.6 and 296.6 \( \mu s \) respectively for default and proposed design. For collect operation, we evaluated with 15 process running on host and MIC, as the collect benchmark is memory intensive. (We plan to present the 16 process per host and MIC results for the camera-ready submission.) At 960 processes, the collect operation latencies observed for 4 KB message is 73726.9 and 39947.4 \( \mu s \), respectively. The barrier performance results presented in Figure 6.13 indicates that the operation latencies for MV2X-Def increases significantly with increase in scale. However, MV2X-Proposed offers much lesser latencies, and the results indicate good scalability. The barrier latency observed at 1,024 processes are 2498.6 and 216.0 \( \mu s \) respectively for MV2X-Def and MV2X-Proposed, which is 11X reduction in latency.
Figure 6.10: Broadcast Latency

Figure 6.11: Reduce Latency

Figure 6.12: Collect Latency
6.4.4 Symmetric Mode Performance Evaluation

As indicated in Section 1, a parallel job can be run in symmetric mode (processes on host and MIC). In this section, we evaluate the performance of applications running in symmetric mode. We evaluate this mode using Graph500, with 16 processes running on each host and MIC. We varied the system scale from 128 processes to 1,024 processes. We used a graph with four million vertices, as input for this evaluation. The performance results are presented in Figure 6.14(a). As it can be observed from the figure, the proposed design improves the Graph500 performance significantly, and for all different scales. Note that the improvement increases with increase in scale, indicating good scalability characteristics. At 1,024 processes, the execution time reported are 15.9 and 12.4 seconds respectively, which is 28% improvement.
6.4.5 Native Mode Performance Evaluation

In this section, we present the performance evaluation results for native mode. We use Graph500 benchmark for our evaluations, for varying system sizes from 64 to 512, with each node running 8 MIC processes. We compare the performance of Graph500 benchmark run using MV2X-Def and our design. The execution time of Graph500 BFS kernel is presented in Figure 6.14(b). Since MIC cores have limited memory per core, we use a smaller input size (#vertices = 262,144). As it can be observed from the figure, the proposed design provides lesser execution time for varying system sizes. Note that even though we observed huge improvements for point-to-point and collective benchmarks, the improvement scale is lesser. The main reason here is that, the message size used in Graph500 is much smaller, and for smaller message sizes, the performance of MV2X-Default and MV2X-Proposed is similar. Moreover, since the parallel job contains only MIC processes, several collective designs such as staged collectives [45] cannot be used here. Further, for both the designs, we note that the execution time increases with increase in system size, which indicates scalability limitations on native mode with Graph500 benchmark. However, the application results indicate that the proposed version provides slightly better
performance for all scales. At 512 processes, the execution times reported are 5.17 and 4.9 seconds, for MV2X-Def and MV2X-Proposed designs.

6.4.6 Evaluation of Graph500 with Extensions

We present the performance evaluation results of Graph500 redesigned to use the proposed extensions in this section. We first compare the performance of the new design with host-only mode for varying system sizes, and then compare with symmetric mode as well. Figure 6.14(c) presents the performance evaluation results of the new design compared to host-only mode. The input graph used for this evaluation consists of 16 million vertices. As it can be observed from the figure, the new design improves the execution time for all the different system scales. Further, it is to be noted that the improvement increases with increase in system scale. At 1,024 processes, the execution times reported are 0.33 and 0.26 seconds, respectively for host only version and the new design, which is a 26% improvement in performance.

Additionally, note that the execution times for native and symmetric modes are significantly higher than the new design, and the host-only version. The results indicate that the redesign using the proposed extensions provide magnitudes of performance improvements. The non-global memory allocation allows receive buffers to be allocated only on the MIC processes. While the host processes are sending the vertices, threads in MIC process the received edge information and prepare the newQ to be used for next level. The execution times reported for the default version running in symmetric mode and the re-designed versions are 12.1 and 0.16 seconds, respectively. This is nearly 75 X improvement in performance. These results substantiate the need for extending OpenSHMEM memory model.

161
to support non-uniform memory allocations, and validate our designs. We plan to conduct this evaluation for varying system sizes for the camera-ready submission.

6.5 Summary

We proposed a novel high performance design for OpenSHMEM on Intel MIC architecture. We proposed extensions for allowing non-uniform memory allocations, which are suitable for upcoming heterogeneous memory architectures. We presented proxy-based runtime designs for providing low latency and high bandwidth for communication operations, by efficiently by-passing high overhead communication paths. Our benchmark evaluations reveal 4X-6X reduction in operation latencies. Application level evaluations indicate 28% improvement over the default designs. Further, the application re-designs using the proposed design provides magnitudes of performance improvement compared to the default symmetric mode. To the best of our knowledge, this is the first work, that proposes a high-performance communication runtime for OpenSHMEM for MIC clusters.
Chapter 7: High Performance and Scalable RDMA-based design of Memcached

In this section, we present the proposed designs for extending UCR for supporting middleware for Big Data computing. We also discuss the high performance RDMA-aware design of Memcached using UCR. Further, we present the proposed scalability enhancements in UCR by making use of hybrid transport protocols.

7.1 Designing Unified Communication Runtime (UCR) for Data-Center Environments

The Unified Communication Runtime (UCR) aims to unify the communication runtime requirements of scientific parallel programming models, such as MPI and Partitioned Global Address Space (PGAS) along with those of data-center middleware, such as Memcached. Our approach is to re-use the best communication runtime designs across these domains.

The requirements imposed by these disparate models are quite challenging. For example, in the parallel computing domain, there is a notion of a parallel “job.” Processes belonging to a job may communicate with others using a rank or a thread id (in case of PGAS). This is quite different from the Memcached model, where clients maintain a list, or pool, of available Memcached servers. Based on the data they want to access, keys
Figure 7.1: Active Messaging in UCR
are assigned to the data and a server is chosen using a hash function. Another important
distinction is that of fault tolerance. In MPI or PGAS, when a process belonging to a job
unexpectedly fails, the entire job fails. However, in the data-center domain, failure of one
Memcached server or client must be tolerated.

There are several new contributions to UCR made as a part of this work. They include
a fault-tolerant model (one failing process doesn’t fail others), a revamped active message
API, and active message counters. They are described in detail below:

### 7.1.1 Communication Model and Connection Establishment

As discussed above, the communication model for Memcached does not allow for one
failing process to impact others. Based on this requirement, we have re-designed the con-
nection establishment process in UCR. Previously, UCR only accepted destination ids (such
as ranks in MPI or thread ids in PGAS). The new version of UCR follows an end-point
model. The client must establish an end-point with the server before communication be-
gins. Each end-point is bi-directional, i.e. the server can talk back to the client using the
same end-point. The client has a choice of the type of end-point that can be used (reliable
vs. unreliable). This is similar to either TCP or UDP sockets.

Another addition to the UCR API is that of synchronization with timeouts. In MPI
or PGAS, an erroneous process (either software or hardware failure) can result in the job
hanging. This is not permitted in the data-center environment. Accordingly, we have in-
troduced UCR methods to wait for events, but with specified timeout values. If events do
not occur within that time limit, then Memcached can choose to take corrective action. For
example, a client may decide that a server has gone down.
7.1.2 Active Messaging in UCR

Active messages were first introduced by Von Eicken et. al. [82]. They have proven to be very useful in many environments, such as PGAS library design, e.g. GASNet project from U. C. Berkeley [22] and MPI design using the Low-Level API (LAPI) from IBM [33]. They were never considered in the data-center environment in the past. To the best of our knowledge, this is the first application of Active Messaging in commodity RDMA capable networks in the data-center domain.

UCR provides interfaces for Active Messages as well as one-sided put/get operations. The basic operation of Active Messages in UCR is illustrated in Figure 7.1. The process that initiates communication, is called the Origin, the destination is called the Target. An example of the general working of active messages in UCR is shown in Figure 7.1(a). An active message consists of two parts: header and data. When an active message arrives at the target, a ‘header handler’ is run. The header handler is a short function that can perform some limited logic and identify the destination buffer of the data. Then, UCR can transfer the data to the identified location using RDMA operations. Once the data has been put, a ‘completion handler’ is run at the target, which can perform any operation on the data. It should be noted that the choice to run completion handler is also optional and depends on the logic in the header handler. The associated counters and ‘optional messages’ are explained in the following section.

As an optimization for short messages, header and data may be combined into one network transaction. In that case, instead of RDMA operation, the data can be simply copied off the network buffer into destination location. An example of the active Message API in UCR is as shown below:
int ucr_send_message( ucr_ep_t *ep, int msg_id, void *hdr, size_t hdr_len, void *data, size_t data_len, ucr_cntr_t *origin_counter, ucr_cntr_t *target_counter, ucr_cntr_t *completion_counter )

Arguments: ep identifies the target, msg_id identifies the header handler to be run on the target, hdr, hdr_len, data, data_len indicate the header and data to be sent. The counters are explained in the following section.

7.1.3 Active Message Counters

Counters provide a generalized and clean way to track progress of active messages. Counters are objects with monotonically increasing values. There are three types of counters: origin_counter, target_counter and completion_counter. origin_counter and completion_counter belongs to the origin, whereas the target_counter belongs to the target.

**Origin counter:** it is incremented at the origin when the memory containing header and data can be re-used by the application (e.g. Memcached). In the case of large active messages, once the RDMA of the data is complete, a message is sent to the initiator side that results in the counter being updated. This is shown in Figure 7.1(a). If the origin counter supplied to ucr_send_message is NULL, then this extra message is not issued by UCR. For short messages, UCR can just look for local completion and update the counter. Local completion here means, when the network layers indicate that buffer can be safely re-used. This may mean that the message is internally copied into a buffer. However, it is safe to re-use the application level buffer. **Target counter:** it is incremented at the target when the data has completely arrived and completion handlers have run. **Completion counter:** it is incremented at the origin when the completion handler at the target side is complete.
Note that if the supplied value of completion_counter is NULL, then UCR will not issue the optional internal message.

7.2 RDMA-Aware Memcached Design Using UCR

![Diagram of Memcached Set and Get Operations using UCR]

(a) Set Operation

- **Active Message 1**
  - Client to Server
  - (Large message: depending on item, key K and counter C)

- **Get Data (RDMA)**

- **Result of Set operation**
  - Mark success/failure
  - Target Counter (C) ++

- **Active Message 2**
  - Server to Client
  - (Short message: result of set; Target Counter: C)

(b) Get Operation

- **Active Message 1**
  - Client to Server
  - (Short message: with key K and counter C)

- **Get Request**

- **Get Data (RDMA)**

- **Item info: length of item**

- **Alloc/get memory for length of item**

- **Target Counter (C) ++**

- **Active Message 2**
  - Server to Client
  - (Large message: depending on item; Target Counter: C)

Parse Command, Identify Slab, Item ..

Stored Successfully?
Yes/No

Figure 7.2: Network Transactions for Memcached Set and Get Operations using UCR
In this section, we describe our design of Memcached using the UCR active message APIs in detail. The active message concepts in UCR were explained in the previous section. In this section, we use the active message concepts to illustrate the design of Memcached. We focus on the set and get operations. These are the fundamental operations included in Memcached APIs. Other operations such as: mget, increment, decrement and delete can be implemented using these principles. We used Memcached server version 1.4.5 [7] and client C library (libmemcached) version 0.45 [6].

7.2.1 Connection Establishment

Memcached relies on libevent for detecting sockets which have become ready for communication. Based on which socket is ready for communication, it is handed over to a server thread. We maintain this overall model with some modifications. The goal of our design is to maintain compatibility of the existing Memcached server to work with both Sockets based clients and UCR based clients. In our version of Memcached, the connection establishment process begins as normal. However, the UCR clients indicate UCR compatibility. UCR client request causes a libevent notification at the server side and a worker thread is assigned (in a round-robin manner) for this client. Client and worker thread both create end points and start communication using active messages over UCR. All further requests from client will be served by this worker thread. Note that a worker thread can handle several clients at a time. The number of worker threads can be set using a runtime parameter. The underlying UCR calls are always non-blocking and asynchronous.

7.2.2 Set Operations

Using the set operation, a Memcached client can put an object (item) in Memcached server memory. Each item is assigned a key, which can be a hash value computed from its
contents. However, there is no restriction on how the key is chosen: it is up to the client. Once the key is computed, using another hash operation, a target server is chosen from the pool of available servers. Then, the set operation is issued.

An illustration of the network operations that ensue when a set operation is issued is presented in Figure 7.2(a). First, the client issues an active message to the server with the set request (AM 1). The basic operation of active messages in UCR was described in Section 7.1. This active message can be large (depending on size of item being stored). The client also indicates a counter C. After the server parses the command from the client, it identifies where it wants to store the item. Then, it issues an RDMA Read to that destination memory location. When the store is complete, it indicates the status of the completion back to the client using another active message (AM 2). Meanwhile, the client is waiting on counter C to be incremented. This is a blocking call with client specified timeout. The server indicates this counter C as the “target counter” in AM 2. AM 2 is typically a short message, as it only contains the status. When it completes at the target (which is the Memcached client), counter C is incremented. The Memcached client then knows that the server has responded with an answer. It then inspects the contents of the message received and takes appropriate actions.

### 7.2.3 Get Operations

Using the get operation, a Memcached client can get an object (item) from a Memcached server. The key of the item indicates which server it may be stored in (through a hash function). Once the server is identified, the get operation is issued.

An illustration of the network operations that ensue when a get operation is issued is presented in Figure 7.2(b). As with the case in set, the client indicates the counter C it
is waiting on in the first active message (AM 1). Once the server gets the command from
the client, it finds the requested item. In standard Memcached API, the length of the item
is not known by the client before-hand. It is known only at the server. The server then
responds with active message AM 2 with the Memcached client as the target and counter C
as the “target counter.” Once the client learns the size of the item being sent by the server,
it allocates required memory (it may implement a local buffer pool). Then, the item is read
into the destination buffer. When data is available, UCR increments the target counter, C.
The client then knows that the get operation is complete.

**Note on Small Set/Get operations:** As described in Section 7.1, UCR is optimized for
small data. In the case the amount of data being set or get fits into one network buffer
(8 KB), it is packaged within one transaction. In this case, no RDMA calls are used, and
memcpy is used at the target to copy data into destination buffers.

### 7.2.4 Performance Evaluation

We describe the experimental setup and the results of our performance analysis of Mem-
cached in this section. Our benchmarks are inspired by the popular `memslap` benchmark
that is distributed along with the Memcached client library. This benchmark measures
latency and operations per second for varying mixes of Set and Get operations. Unfor-
tunately, this benchmark does not utilize the `libmemcached` API itself. Rather, it directly
sends messages using Sockets. Therefore, we created our suite of benchmarks that perform
similar evaluation, but use the standard `libmemcached` C API. We used Memcached server
version 1.4.5 and client (`libmemcached`) version 0.45. The client behavior was set using:
```
memcached_behavior_set(memc, MEMCACHED_BEHAVIOR_TCP_NODELAY, 1).
```
This resulted in better and more predictable latency performance.
7.2.4.1 Experimental Setup

We use two generations of multi-core architectures from Intel distributed across two different clusters for our evaluation.

Intel Clovertown Cluster: This cluster consists of 64 compute nodes with Intel Xeon Dual quad-core processor nodes operating at 2.33 GHz with 6 GB RAM, a PCIe 1.1 interface. The nodes are equipped with a ConnectX DDR IB HCA (16 Gbps data rate) as well as a Chelsio T320 10GbE Dual Port Adapter with TCP Offload capabilities. The operating system used is Red Hat Enterprise Linux Server release 5.5 (Tikanga), with kernel version 2.6.30.10 and OpenFabrics version 1.5.1. The IB cards on the nodes are interconnected using a 144 port Silverstorm IB DDR switch, while the 10 GigE cards are connected using a Fulcrum Focalpoint 10 GigE switch. This cluster is about four years old, and represents the “trailing” edge of the technologies used in commodity clusters.

Intel Westmere Cluster: This cluster consists of 144 compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12 GB RAM. Each node is equipped with MT26428 QDR ConnectX HCAs (36 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected using 171-port Mellanox QDR switch. The operating system used is Red Hat Enterprise Linux Server release 5.4 (Tikanga), with kernel version 2.6.18-164.el5 and OpenFabrics version 1.5.1. This cluster is only about a year old and represents the “leading” edge of the technologies used in commodity clusters.

Note Regarding Interpreting Results: It is to be noted that the performance improvements in Memcached performance seen with our UCR based design are mainly due to OS-bypass and the match in memory-oriented nature of RDMA semantics as opposed to stream oriented socket semantics. UCR provides a good design of active messaging on
Figure 7.3: Latency of Set and Get Operations on Intel Clovertown Cluster
top of OpenFabrics verbs. We may expect to see good gains in performance with the iWARP/RoCE implementations of UCR that will run over a 10 GigE network. We are currently in the process of designing the iWARP and RoCE versions of UCR.

For all SDP related experiments zero-copy was turned off as the existing sockets based Memcached code requires sockets to be in non-blocking mode while the default SDP implementation shipped with OFED does not work in non-blocking mode with zero copy [29]. Without turning off zero copy, Memcached with SDP crashes with an error.

7.2.4.2 Performance of Memcached with Single Client

In this section, we look at the single client performance with Memcached for various operations over different network interconnects.

Performance of Set and Get:
The first set of experiments measure the performance for set and get operations. In these experiments, the Memcached client repeatedly sets (or gets) a particular size of item from the cache. i.e. the Memcached instruction mix is either 100% Set or 100% Get.

Performance on Intel Clovertown Cluster: Data for Set and Get operations are shown in Figure 7.3. We observe that UCR based design outperforms 10 GigE with TCP offload by at least a factor of four for all message sizes. We can also see that the UCR based design outperforms SDP and IPoIB by a factor of eight or more for small to medium message sizes and by a factor of five for large message sizes.

Performance on Intel Westmere Cluster: Performance of Set and Get operations are shown in Figure 7.4. We observe that, the UCR based design outperforms SDP and IPoIB based options by at least a factor of ten for all small message sizes and up to a factor of four for large message sizes. Due to lack of 10 GigE cards on this cluster, we were not able to present corresponding 10 GigE numbers. We observed that the SDP results on this
cluster were noisy. We made several attempts to reduce the jitter by increasing the number of samples taken in the experiments (up to 10,000). However, the jitter did not subside. We conclude that this must be an implementation artifact of SDP on QDR adapters. The fact that IPoIB and UCR results are jitter free indicates that it is not a problem of the underlying network itself.

These results clearly underline the performance benefits that we can gain through OS-bypass and memory-oriented communication offered by RDMA semantics. These benefits can be leveraged using UCR.

**Performance of Mixed Set and Get:**

The second set of experiments mix the instruction set of Memcached operations. We use a mix of 10% Set operations and 90% Get operations. The pattern of access is 10 Sets followed by 90 Gets. We call this pattern non-interleaved. We also evaluate an interleaved pattern in which the instruction mix is 50% Set operations and 50% Gets. In this pattern, 1 Set is followed by 1 Get. We restrict the presented data to small messages due to space limitations.

**Performance on Intel Clovertown Cluster:** Data for mixed operations is shown in Figures 7.5(a) and 7.5(c) for the non-interleaved and interleaved instruction mix described above. We can see that these mixed operations follow the same trends as the basic Set and Get operations. We see that the UCR based design outperforms 10 GigE with TCP offload by at least a factor of *four* for all message sizes. We can also see that the UCR based design outperforms SDP and IPoIB by a factor of *seven* or more for small to medium message sizes and by a factor of *four* for large message sizes.

**Performance on Intel Westmere Cluster:** Data is shown in Figures 7.5(b) and 7.5(d). We can see that these mixed operations follow the same trends as the Set and Get operations
Figure 7.4: Latency of Get and Set Operations on Intel Westmere Cluster
seen above. We see that the UCR based design outperforms the SDP and IPoIB based designs by a factor of 10 or more for small to medium message sizes and by a factor of four for large message sizes.

![Graphs showing latency of small messages for Non-Interleaved and Interleaved operations on different clusters.](image)

Figure 7.5: Latency of Small Messages for Non-Interleaved (Set 10% Get 90%) and Interleaved (Set 50% Get 50%) Operations on Intel Clovertown and Intel Westmere Clusters
7.2.4.3 Performance of Memcached with Multiple Clients

In this section, we look at the performance of Memcached with multiple clients over different network interconnects. We restrict our evaluation to Get operations due to space limitations. The multi-client benchmark is an extension of the single client benchmark discussed in Section 7.2.4.2. All the clients are started simultaneously. Instead of latency, we report the total number of transactions that were executed per second. The number reported is the aggregate transactions per second observed by all the clients. This benchmark is expected to simulate the case where many clients access the same Memcached server simultaneously.

We vary the clients from eight to sixteen. The clients are all located on different nodes than the Memcached server. The results for two message sizes (4 Byte and 4KB) are presented in Figure 7.6. On the Clovertown Cluster, we observe that the transactions per second achieved by Memcached over UCR is a factor of six better than that of 10-Gigabit Ethernet with TCP acceleration for small transactions. The 10-Gigabit TOE outperforms SDP over InfiniBand. On the Intel Westmere Cluster, we can observe that the improvements in transactions per second is quite significant, a factor of six over the corresponding SDP version on the Intel Westmere Cluster for 4 Byte messages. The transactions per second is around 1.8 Million operations/sec with UCR on QDR adapters. We observe that the SDP performance is lower than that of IPoIB. This is similar to our latency observations in Figure 7.4. We believe that this could be a software issue with SDP.

7.3 Scalable Memcached Design using hybrid transport protocols

In this section, we present the design and implementation details of Unified Communication Runtime (UCR) over Unreliable Datagram transport (UD). We also describe the
Figure 7.6: Number of Transactions per Second for Get Operation on Intel Clovertown and Intel Westmere Clusters
enhancements made to Memcached in this work and how Memcached design can benefit from UCR-UD.

7.3.1 Enhancing UCR to support Hybrid Transports

As discussed in Section 2.1.1, the memory requirement of RC transport increases as the connections scale up. On the other hand, UD requires almost constant memory resources. However, the UD transport has several limitations (Section 2.1.1). We propose the following solutions to address these challenges and improve the scalability of Memcached.

7.3.1.1 Reliability and Flow Control

With UD transport, it is critical to efficiently handle reliability and flow control. Our UCR-UD design is based on the traditional sliding window protocol. The sender issues packets depending on the space available in the window, which represents the maximum number of unacknowledged message segments. Send operations that may occur when the window is already full are queued until outstanding operations have been acknowledged. We assign a time-stamp to each segment that is sent and if an ACK has not been received within a given timeout, the segment is re-transmitted.

7.3.1.2 Large Message Transfers

RC transport can employ Remote Direct Memory Access (RDMA) efficiently for large message transfers. With UD, messages larger than the MTU size have to be segmented and sent as separate packets and they have to be reassembled at receiver, which involves two memory copies. We employ the Zero Copy Transfer protocol [50] to address these limitations.
7.3.1.3 Hybrid Connection Management

We also consider hybrid connection models [46]. Using this approach, we try to leverage the advantages of both RC and UD. We can limit the maximum number of RC connections to a specified threshold. Further connections can be made over UD transport. This ensures that the memory requirement does not increase above a particular limit. Another possibility is to dynamically change the transport mode from UD-to-RC (upgrade) or RC-to-UD (downgrade), based on the communication frequency between the server and a specific client, and we can also impose a limit on the maximum number of RC connections. These hybrid modes can be selected depending on platform characteristics.

7.3.1.4 Communication Performance

Along with the transport level enhancements, we have also introduced a ‘Registration Cache’ in UCR. For InfiniBand, the communication buffers have to be registered with the adapter. However, the memory registration and de-registration operations are expensive and if they are done for every communication operation, it significantly affects performance. Registration cache allows us to re-use registered memory pages and these are deregistered only when they are evicted from the cache.

7.3.2 Memcached based on UCR-UD

We have designed the UCR-UD interface to allow Memcached to transparently use the same set of API’s. The connection transport type is identified by the `transport_type` argument, during the creation of the communication end-point. Our design guarantees that Memcached can conveniently leverage our work without any changes. Hybrid configuration parameters can be set using run-time environment variables. UCR transparently switches between transport protocols in hybrid mode for best performance. We have also
made several enhancements over the Memcached design presented in [41]. These include support for multiple Memcached servers and handling of Memcached operations such as ‘delete’ and ‘flush’. We use the multi-server configuration in one of the application benchmarks. Detailed description and results are presented in Section 7.3.2.1.7.

7.3.2.1 Performance Evaluation

We describe the experimental setup and the results of our performance analysis of Memcached in this section. We present micro-benchmark and application-benchmark level analysis as well as results based on real application workloads. The micro-benchmarks used in experiments are based on the popular memslap benchmark distributed along with Memcached client library. The application benchmark used is Apache Olio [15] benchmark. We compare the performance of Memcached design over UCR (UD and RC) with that of the socket variants (SDP, IPoIB, 1GigE and 10GigE). We used Memcached server [7] version 1.4.9 and client (libmemcached) [6] version 0.52. For socket versions, the client behavior was set using MEMCACHED_BEHAVIOR_TCP_NODELAY. This resulted in better and more predictable latency performance. The clusters which we used for our experiments have 8 CPU cores per node. Memcached Server was configured with 8 ‘worker threads’ for better utilization of CPU cores.

7.3.2.1.1 Experimental Setup

We used two generations of multi-core architectures from Intel distributed across two different clusters for our evaluation.

Intel Westmere Cluster (Cluster A): This cluster consists of 144 compute nodes with Intel Xeon Dual quad-core processor nodes, operating at 2.67 GHz. Each node has 12 GB of memory and is equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected using 171-port Mellanox
QDR switch. The operating system used is Red Hat Enterprise Linux Server release 5.4 (Tikanga), with kernel version 2.6.18-164.el5 and OpenFabrics version 1.5.1. This cluster represents the leading edge of the technologies used in commodity clusters. It is termed as “Cluster A” in experiment results.

**Intel Clovertown Cluster (Cluster B):** This cluster consists of 64 compute nodes with Intel Xeon Dual quad-core processor nodes operating at 2.33 GHz with 6 GB RAM, a PCIe 1.1 interface. The nodes are equipped with a ConnectX DDR IB HCA (16 Gbps data rate) as well as a Chelsio T320 10GbE Dual Port Adapter with TCP Offload capabilities. The operating system used is Red Hat Enterprise Linux Server release 5.5 (Tikanga), with kernel version 2.6.30.10 and OpenFabrics version 1.5.1. The IB cards on the nodes are interconnected using a 144 port Silverstorm IB DDR switch, while the 10 GigE cards are connected using a Fulcrum Focalpoint 10 GigE switch. This cluster represents the trailing edge of the technologies used in commodity clusters. This cluster is termed as “ClusterB” in experiment results.

### 7.3.2.1.2 Microbenchmark Analysis

We present the performance results of Memcached Set/Get operations, transaction throughput evaluation and scalability analysis in this section.

### 7.3.2.1.3 Performance of Memcached Set/Get operations

This set of experiments measure the performance of the key Memcached operations, *Set* and *Get*. In the experiment, Memcached client repeatedly issues *Set* (or *Get*) operations for a fixed number of iterations. We repeated this for different message sizes. The average operation latency is reported in the results. We did this experiment for different *Set/Get* mixtures: 100% *Set,*
Figure 7.7: Latency of Memcached Operations - Cluster A
100% Get and 50% Set - 50% Get. We performed the evaluation on clusters A and B and results are shown in Figure 7.7 and Figure 7.8, respectively.

We compare the performance of Memcached design using UCR (both RC and UD transports) with that of pure Memcached implementation on various socket protocols. Cluster A is configured with the following socket protocols - SDP, IPoIB, 1GigE, and 10GigE. Figure 7.7(a) and Figure 7.7(b) depict the performance of Memcached Get operations on Cluster A. We have split the results into two graphs for fine grained analysis based on message size. As it can be observed from the figure, Memcached over UCR achieves lowest latency. For 4 byte message size, the Get operation latency is just about 4.28 $\mu$s and 4.86 $\mu$s for Memcached design over UCR-RC and UCR-UD, respectively. The operation latency of the best performing socket version, SDP is 54.32 $\mu$s. This is a factor of 12 X higher than the latency of Memcached over UCR. The performance gain exists for larger message sizes also. The operation latencies of Set operation and mixed Get/Set operations are presented in Figures 7.7(c) - 7.7(f). The performance patterns of these experiments are similar - Memcached over UCR achieves the lowest latency.

Figure 7.8 depicts the results in Cluster B. Since this cluster contains 10 GigE cards, we present Memcached performance results over 10 GigE also. For 4 byte message size, the Get operation latency is just about 6.82 $\mu$s and 7.73 $\mu$s for Memcached design over UCR-RC and UCR-UD, respectively. The operation latency of the best performing socket version, 10 GigE is 48.86 $\mu$s. This is a factor of 7 X higher than the latency of UCR versions. The latency results of UCR-RC and UCR-UD are a little higher than those observed in Cluster A. This is because Cluster B contains a slower InfiniBand adapter (16 Gbps) as compared to Cluster A (32 Gbps).
It is to be noted that the performance of UCR-UD is a little lower as compared to UCR-RC in both the clusters. RC transport offers built-in reliability, which is implemented in hardware layer. But, UD transport does not guarantee reliability and flow control, and these mechanisms shall be implemented in the software layer. Because of these software overheads, UD performance is a little lower than RC. Moreover, messages larger than MTU size (2 KB) have to be segmented and shall be sent in chunks. On the other hand, RC transport offers high performance features such as RDMA for large message transfers.

7.3.2.1.4 Transaction Throughput  Memcached transaction throughput indicates the maximum number of Get/Set operations that the server can handle per second. In the experiment, we used one Memcached server and varied the number of clients from 1 to 1,024. We used Get of 4 bytes as the operation. We ran this experiment on Cluster A using Memcached over UCR (RC and UD) and Memcached over socket interfaces (SDP, IPoIB and 1 GigE). Results are shown in Figure 7.9(a). As it can be observed from the figure, Memcached over UCR delivers higher throughput. Memcached over UCR-RC and UCR-UD provide peak throughput of 1.4 M operations per second and 1.3 M operations per second, respectively. This is 3 X times higher than that of Memcached over SDP.

It is interesting to note that the transaction throughput of Memcached over UCR increases and reaches a peak at 8 clients and then stabilizes to around 750 K operations per second. Cluster A has compute nodes with 8 CPU cores. We configured Memcached server (both UCR and socket versions) with 8 worker threads, so that all the cores are utilized efficiently. This is the reason we obtain peak throughput at 8 clients. When the number of clients is increased further, each worker thread serves more than one client and CPU performance becomes the bottleneck. Even though network offers more bandwidth, CPU
Figure 7.8: Latency of Memcached Operations - Cluster B
performance bottlenecks the throughput. This trend is not visible in socket versions because network bandwidth is the bottleneck for these. Similar characteristics were observed in previous Memcached performance analysis [55].

7.3.2.1.5 Scalability Analysis In this section we analyze the memory requirement of Memcached server, when scaled up. We measured the memory footprint for varying number of clients connections - 1 to 4 K. The results are presented in Figure 7.9(b). As it can be observed from the figure, the socket based versions have higher memory footprint, around 630 MB. Socket based design keeps per-thread shared connection buffer pool for TCP and UDP sockets. Each buffer pool is about 64 MB, and the experiments were run using 8 worker thread configuration. It is to be noted that keeping per thread buffer pool might not be efficient especially when multi/many core clusters are evolving.

InfiniBand RC transport needs to create QP for each client connection (Section 2.1.1). Size of each QP depends on the queue size. We used a typical configuration of 64 ‘send work queue’ elements and 128 ‘receive work queue’ elements. As the number of client connections increase, memory footprint of Memcached server with UCR-RC increases. For 4 K client connections, it almost doubles the initial memory footprint. For UD transport, there is no need to create new QP for each connection. As a result, it keeps a steady memory footprint size. This behavior is favorable for applications like Memcached. If there is an increase in memory requirement as the connections increase, it can adversely affect the memtable size and degrade the performance when scaled up. The initial memory footprint (when number of client connections is zero) of UCR-UD is a little higher as compared to UCR-RC. This is because more receive buffers are posted in UCR-UD for handling connections at scale.
We ran the experiment for hybrid connection model as well. In this model, we limit the maximum number of RC connections to 512. It can be observed that the memory footprint of UCR-Hybrid remains steady after 512 client connections. Socket based versions keep a steady memory footprint, but the performance is not at all comparable to UCR based variants.

Figure 7.9: Scaling Memcached - Transaction Throughput and Memory Footprint

### 7.3.2.1.6 Apache Olio Benchmark Results

We evaluated our design using application benchmarks as well. We used Apache Olio benchmark [15] and simulated the workload using Faban [26]. Olio is a social web calendar application. A detailed description of Olio Benchmark and Faban Workload Generator is given in Section 2.6.1.

In our experiment, we used the workload for 25,000 application users per Memcached client. We varied the number of Memcached clients from 1 to 1,024 and the average time...
for handling workloads per user is reported. The workload contains a mixture of Memcached Set and Get operations (90% Gets and 10% Sets). The data size of these operations vary in the range of 4 bytes to 32 Kbytes.

![Graphs](image1.png)

(a) Comparison with socket-based designs

(b) Comparison with RC Transport

Figure 7.10: Apache Olio Benchmark Results

Results of this experiment are shown in Figure 7.10. Figure 7.10(a) compares the performance of Memcached over UCR with that of the socket variants. As it can be observed from the figure, Memcached-UCR over RC and UD performs significantly better than the socket versions. For 8 client (25,000 x 8 users) benchmark, UCR-RC and UCR-UD takes around 12.95 ms and 17.52 ms for complete. This is a factor of four better than the IPoIB results. It is to be noted that even though SDP performed better than IPoIB in microbenchmarks, it does not hold true in case of application results. For 8 client benchmark, Memcached over SDP took 99.94 ms whereas IPoIB took 48.13 ms. This holds true for other application benchmarks also, as discussed in Section 7.3.2.1.7.

We performed the experiment for higher scale also. These results are depicted in Figure 7.10(b). In order to do a fine grain analysis of UCR-RC and UCR-UD, we did not
include the results of socket based variants. Since the workload contains large data sizes, there is a visible performance difference between UCR-UD and UCR-RC results. This is because UCR-RC employs RDMA for large message transfer. Since RDMA support is not there in UD transport, messages are segmented and sent in 2 KB packets.

We used UCR in a hybrid connection mode as well, in which some connections use RC transport and some other use UD. We limited the maximum number of RC connections to 512. This can be set as a runtime parameter and it can be tuned as per system/application characteristics. The hybrid mode results are indicated as ‘UCR-Hybrid’ in the results. It can be noted that UCR-RC and UCR-Hybrid results are almost same for less than 512 client connections. For 1, 204 clients, UCR-RC took 1651.56 ms, whereas for hybrid mode took 1701.35 ms. With hybrid mode we achieve comparable performance to that of RC, while keeping a limit on memory footprint.

7.3.2.1.7 Results based on Real Application Workloads In order to further evaluate our design, we ran experiments based on real application workloads. Our benchmarks are based on Internet workloads handling user actions. User actions on the Internet are typically kept lightweight, in order to avoid slow browsing experience. These generate condensed logs on the server side with information such as browser cookies, session cookies and other such concise identifiers. Processing of these user action logs involves attribution/lookups with the actual complete information needed for further action. Such functions are often batched and processed in small chunks. With current Internet scales, such lookup data can be very large and performance requirements are quite high.

In our experiment, we used the following configuration. Each Memcached client issues 0.5 million Get operations followed by Set operations which update 20% of data set. The
value length varies from 1 KB to 4 KB and key length varies from 3 to 20 bytes. These benchmarks exhibit ‘weak scaling’ characteristics. We increased the total data size, with increase in number of clients. Each client adds 2,000 key-value pairs. The entire data set was cached using three Memcached servers. Read and update operations were repeated over several iterations and the average time is reported.

Figure 7.11(a) compares the performance of Memcached over UCR with that of socket based variants. The performance characteristics are similar to that of Olio Benchmark results. Memcached over UCR performs significantly better than other socket based variants. For 8 clients, UCR-RC and UCR-UD take 6.1 ms and 6.2 ms, respectively. This is about 12 X times better than IPoIB performance (72.6 ms). As observed in Olio results, SDP does not perform as good as IPoIB. Figure 7.11(b) depicts the performance of Memcached over UCR at scale. Since the data size is comparatively smaller to that of Olio, the performance of UCR-RC and UCR-UD are almost similar. UCR-Hybrid (number of RC connections capped to 512) results are also presented. For 1024 clients, the execution times using UCR-RC, UCR-UD and UCR-Hybrid were 302.18, 318.04 and 314.93 ms, respectively.

Figure 7.11: Real Application Workload Results
7.4 Summary

We presented a novel design of Memcached for RDMA capable networks. Our design extended the existing open-source Memcached software and made it RDMA capable. We provided design details of our Memcached implementations underlying communication layer – Unified Communication Runtime (UCR). UCR provides an easy to use Active Message API that matches very well with requirements of Memcached. We have significantly enhanced our UCR design to suite a wide range of needs from high performance communication runtimes. We provided a detailed performance comparison of our Memcached design compared to unmodified Memcached using Sockets over RDMA and 10 Gigabit Ethernet networks with hardware-accelerated TCP/IP. Our performance evaluation shows that latency of Memcached Get of 4KB size can be brought down to 12 µs using ConnectX InfiniBand QDR adapters. Latency of the same operation using older generation DDR adapters is about 20 µs. These numbers are about a factor of four better than the performance using 10 GigE with TCP Offload. In addition, these latencies of Get requests over a range of message sizes are better by a factor of five to ten compared to IP over InfiniBand and Sockets Direct Protocol over InfiniBand. Further, throughput of small Get operations can be improved by a factor of six when compared to Sockets over 10 Gigabit Ethernet network. Similar factor of six improvement in throughput is observed over Sockets Direct Protocol using ConnectX QDR adapters.

We further enhanced UCR to using hybrid InfiniBand transports. We implemented the hybrid RC/UD transport, and designed Memcached using UCR. We addressed several issues with the UD transport including reliability, flow control and communication performance. We also proposed schemes that allow connections to transparently switch from UD and RC to offer a hybrid transport to the Memcached stack. Through our work, we could
achieve a good balance of performance and scalability by using efficient hybrid schemes. Our experimental evaluations revealed that our design is highly scalable, with comparable performance as that of RC. Memcached Get latency for 4 byte data size is 4.28\(\mu s\) and 4.86\(\mu s\) for RC and UD transports, respectively. Latency results for 4 KB data size is about 9\(\mu s\) and 10\(\mu s\) for RC and UD transports, respectively. This is a factor of 6 X to 12 X better than SDP performance. In evaluations using Apache Olio benchmark with 1,024 clients, Memcached execution time using RC, UD and hybrid transports are 1.61, 1.96 and 1.70 seconds, respectively.
Chapter 8: High Performance RDMA-based design of HBase

We present our proposed RDMA-based design of HBase using UCR. Unlike Memcached, HBase is implemented in Java and requires efficient interface to communicate using the C-based UCR. We describe the design details and performance evaluations in this section.

8.1 Detailed Design

We present our hybrid design for HBase to incorporate both socket and advanced networks such as InfiniBand in this section. First, we briefly walk through the existing socket-based HBase communication flow. Then we describe the new hybrid architecture and explain its key components. Our design extends HBase communication management policies to provide hybrid communication support. It makes use of UCR for InfiniBand communication via JNI Adaptive Interface.

8.1.1 Socket-based HBase Communication Flow

In our hybrid framework, the original components in HBase software stack are kept intact to support the conventional sockets interface. We briefly outline some key components in HBase before explaining our design.
Figure 8.1: Architecture and Workflow of HBase Design with Hybrid Approach

On HBaseRegionServer, there is a **Listener** thread to monitor Java Sockets. When incoming data is detected, the Listener picks a thread from the **Reader Thread Pool** to process the data. Once a **Reader Thread** is awaken by the **Listener**, it reads data from the socket, de-serializes the data into a Java **Call** object defined in HBase, and pushes it into a queue associated with a **Handler Thread Pool** (path 7 in Figure 8.1). One of the **Handler Threads** claims the **Call** object and performs the actual processing. Once the processing is completed, the **Handler Thread** puts the **Call** object together with the serialized reply data into the response queue which is maintained by the **Responder Thread** (path 8 in Figure 8.1). The **Responder Thread** sends the reply data back to the client (path 9 in Figure 8.1).
8.1.2 Hybrid Communication Framework to Support High Performance Networks

We adopt a hybrid approach by extending the existing HBase communication framework to support InfiniBand and RDMA capability using the UCR library. As shown in Figure 8.1, the new design includes two parts: the HBaseClient side, and the HRegionServer side. We introduce four new components in HBase client side: (1) Network Selector assists in selecting networks according to operation types and pre-defined rules; (2) IB Reader, is responsible for receiving data sent from the HRegionServer and de-serializing into Java objects; (3) JNI Adaptive Interface, enables upper-layer Java code to invoke our native UCR library; and (4) UCR enables RDMA-based data communication over InfiniBand network. On the HRegionServer side, Helper Thread Pool co-operates with the UCR for data communication. Since the connection management is different between HBaseClient and HRegionServer, we chose to implement the JNI Adaptive Interface separately on both sides to operate in a more efficient manner. We discuss these components in detail in the following sections.

8.1.3 Communication Flow over InfiniBand via UCR library

Figure 8.1 highlights the key components involved in data communication. Network Selector enables the hybrid communication mode. It assigns operations to socket helper threads or UCR helper threads based on the operation type. Put and Get operations are assigned to UCR threads, where as all other operations are assigned to socket threads. For clarity, we explain the communication work flow of HBase Get operation here. Application calls the HBaseClient Get API (path 1 in Figure 8.1). Network Selector then passes the Get operation (path 12 in Figure 8.1) to the JNI Adaptive Interface. This interface is the glue
between HBase and UCR, it not only enables HBase to access the UCR, but also manages memory sharing between Java and C. JNI Adaptive Interface invokes UCR communication API (path 13 in Figure 8.1) to send the request via InfiniBand.

UCR is an end-point based communication library. End point is analogous to sockets; client and server use this as a communication end-point. In HBase, clients create end-point and connect to the region server using this end-point. When an operation is issued, it first checks if there is an end-point already with the respective HRegionServer. If end-point exists, it is re-used; otherwise, a new end-point is created. New end-point requests are assigned to Helper Threads in a round-robin manner. Helper Threads deal only with InfiniBand communications, just as Reader Threads deal only with sockets. When a Helper Thread receives a message on one of its assigned end-points (path 18 in Figure 8.1), it de-serializes the request into a Java Call object and puts it into a call queue (path 19 in Figure 8.1) for further processing. This is similar to what the Reader Threads do in case of sockets. In the original HBase design, after a Handler thread finishes processing a request, it pushes the Call object into a response queue, from where a Responder thread writes it back out to the HBaseClient via sockets. In the IB module, the Handler Thread just sets the corresponding end-point’s response status (path 20 in Figure 8.1).

8.1.4 Connection Management

Extending HBase Connection Management: HBase uses Connection object to represent a communication channel between HBaseClient and HRegionServer. Multiple threads multiplex a socket in the Connection to talk with the other side. We extended the HBase connection management to encapsulate both conventional socket interface and the end-point based high performance networks at the same time. The HBase Connection
object now encloses a mapping to keep track of all active end-points as a resource pool. When communication is needed to the other side of the `Connection`, a usable end-point is selected from the pool to perform the data transfer. This end-point pool improves the connection resource usage by multiplexing end-points among many concurrent threads. It also ensures that a request can be issued without any delay.

**Multi-threading Support in HBaseClient:** Most of the practical applications deploy multi-threaded design, to increase concurrency and achieve maximum throughput. YCSB benchmark used in our experiments launches multiple clients as multiple threads. threads for its workloads. As we discussed in previous section 8.1.4, a `Connection` object is created in HBaseClient when it establishes a connection with the target HRegionServer. Each `Connection` object maintains a hash table to store all the requests (`Call` objects) issued by applications. For the single HRegionServer case, all the threads in the application layer share the same connection since their target HRegionServer addresses are the same. Also, in a typical deployment scenario, an HBaseClient needs to communicate with multiple HRegionServers. Therefore, HBaseClient needs to build multiple connections with multiple HRegionServers at the same time. Our HBase design with RDMA over InfiniBand takes this also into consideration. In our JNI Adaptive Interface library, a list of end-points could be created for each connection. Each time, when a thread in the application issues a request, it can get a free end-point to send its request to the target HRegionServer. If there is no available end-point for that connection, a new end-point is created and added into the end-point list. When the request gets reply from HRegionServer, the end-point will be returned back to the end-point list for future use. In the unmodified HBase software, when the `Call` object is sent out to HRegionServer, `call.wait()` is called until client receives data from HRegionServer and finishes the de-serialization of the returned data.
If the returned data size is too large, the overhead caused by the de-serialization could be considerable, and this could block other threads, if there are multiple threads sharing the same connection. In our design, we removed the de-serialization part from the critical path. Once the IB Reader thread gets notified that one end-point’s result is back, it will notify the corresponding Call object to get the returned data in the registered buffer.

8.1.5 Communication Buffer Management

In HBase, Key/Value pairs are organized as Java objects, while the underlying RDMA layer provides memory-based semantics. To bridge the gap between the upper Java object semantics and underlying memory-based semantics, we used Java direct byte buffer in our design. This enables our design to take advantage of zero-copy techniques offered by lower layer communication library.

As we discussed in Section 8.1.4, each connection handles a list of end-points on HBaseClient side, and all these end-points in the same connection share the same chunk of registered buffer for keeping low memory footprint. All the end-points share the same registered buffer chunk and this avoids memory footprint explosion. To avoid interference among these end-points, each end-point maintains a dynamic pointer and size to keep track of the specific region in the registered buffer currently associated with the end-point. Buffer management scheme is the same in both client and server sides.

8.2 Performance Evaluation

In this section, we present the detailed performance evaluation results of our design, as compared to the traditional socket based design over 1 GigE, IPoIB and 10 GigE networks. We conduct micro-benchmark level experiments as well as YCSB benchmark experiments in our evaluations.
(1) Micro-benchmark Experiments: In this set of experiments, we evaluate the latency of HBase `Put` and `Get` operations. We also present detailed profiling analysis of HBase `Put/Get` operations and identify different factors contributing to the overall latency.

(2) Synthetic Workload using YCSB (Single server - Multi-clients): In this experiment, we keep one HRegionServer and vary the number of HBase clients. We use 1 KB payload for the multi-client experiments. This is the default payload in YCSB benchmark.

(3) Synthetic Workload using YCSB (Multi-servers - Multi-clients): In this experiment, we deploy multiple HRegionServers and vary number of HBase clients. This workload is similar to real HBase deployment workloads.

### 8.2.1 Experimental Setup

We use two different clusters in our evaluations.

**Intel Clovertown Cluster (Cluster A):** This cluster consists of 64 compute nodes with Intel Xeon Dual quad-core processor nodes operating at 2.33 GHz with 6 GB RAM. Each node is equipped with a ConnectX DDR IB HCA (16 Gbps data rate) as well as a Chelsio T320 10Gbe Dual Port Adapter with TCP Offload capabilities. Each node runs Red Hat Enterprise Linux Server release 5.5 (Tikanga), with kernel version 2.6.30.10 and OpenFabrics version 1.5.3. The IB cards on the nodes are interconnected using a 144 port Silverstorm IB DDR switch, while the 10GigE cards are connected using a Fulcrum Focalpoint 10GigE switch.

**Intel Westmere Cluster (Cluster B):** This cluster consists of 144 compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12 GB RAM. Each node is equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected
Figure 8.2: HBase Put/Get Latency (Server, Single Client)
using 171-port Mellanox QDR switch. Each node runs Enterprise Linux Server release 6.1 (Santiago) at kernel version 2.6.32-131 with OpenFabrics version 1.5.3. This cluster is less than two years old and it represents the leading edge technologies used in commodity clusters.

We use HBase version 0.90.3 and Hadoop version 0.20.2 in all our experiments. We use the following configuration: Master node and HDFS name-node run on one compute node, while HBase clients and HRegionServers run on different nodes. In this work we focus on the performance potentials of high performance network to accelerate HBase data transmission. So we choose a small data set size in the experiments such that all data to be accessed is completely stored in server memory. By doing this we preclude the possible performance anomalies caused by accessing data from disks. Nowadays, it is common that a server is equipped with 64 GB or more main memory, and the aggregated memory size of many HBase servers is big enough to hold majority of the working set data [67]. Therefore we feel our assumption to cache data in memory is fair, and it can capture the essence of a production HBase deployment. In our future study we will expand our investigations to take disk access cost also into account.

8.2.2 Micro-benchmark Evaluations

We design micro-benchmarks to evaluate the latency of HBase Put and Get operations. The benchmark issues Put or Get operations of a specified size and measures the total time taken for the operation to complete. We run this micro-benchmark in single region server - single client configuration. We keep only one record in the region server, so that all the accesses are serviced from server’s memory store and no disk accesses are involved. The experiment is performed on both Cluster A and Cluster B. We report the results of record
sizes from 1 byte to 64 KB. It is to be noted that, most of the Put/Get payload sizes in a typical Cloud application fall within this range [21]. We also present a detailed profiling analysis of both HBase client and HRegionServer during a Put/Get operation. It provides insights to what all factors contribute to the overall operation latency. We did the profiling for multiple networks and the results are shown in Section 8.2.3.

**Results on Cluster A:** As we can observe from Figure 8.2 that, the UCR based design (denoted as UCR-IB) outperforms the socket based design for all the data sizes. For 1 KB Put operation, the latencies observed for IPoIB, 1 GigE and 10 GigE are 204.4 µs, 269.1 µs and 191.4 µs, respectively. Our design achieves a latency of 86.8 µs, which is 2.2 times faster than 10 GigE. The same level of improvements is observed for Get operations as well. This demonstrates the capability of native InfiniBand RDMA to achieve low latency.

**Results on Cluster B:** The same experiment is repeated on Cluster B and the results are shown in Figures 8.2(c) and 8.2(d). Cluster B does not have 10 GigE network cards. For both Get and Put operations, UCR based design outperforms socket-based channels by a large margin. For Get operation of 1 KB message size, latency observed with our design is 43.7 µs, whereas the latencies are 185.8 µs and 214.4 µs for IPoIB and 1 GigE, respectively. This is a factor of four improvement. For Put operations, the performance improvement over IPoIB and 1 GigE networks is 2.9 X and 3.2 X times, respectively. It is to be noted that UCR based design in Cluster B is faster than in Cluster A, and this is due to the higher InfiniBand card speed (32 Gbps in Cluster B vs 16 Gbps in Cluster A).

These results, illustrated in Figure 8.2 clearly underline the performance benefits that we can gain through OS-bypass and memory-oriented communication offered by RDMA semantics.
8.2.3 Detailed Profiling Analysis

We have profiled HBase client and HRegionServer to measure the cost at different steps during a Put/Get operation. The profiling results, shown in Figure 8.3, indicate the time taken at different steps for Put/Get operation with 1 KB payload.

During a Put operation, the client first serializes the request and puts it into a communication buffer. These are indicated as “Client Serialization” and “Communication Preparation,” respectively. The serialized object is then sent to the server, where it is de-serialized and processed. This step is indicated as “Server Processing.” After handling the request, the server sends back the response to the client. The time taken for serializing the response is indicated as “Server Serialization.” The Client receives the response and processes it (indicated as “Client Processing”). The overall communication time (client to server and server to client) is denoted as “Communication Time.” The actual number of bytes written on wire for Put operation of 1 KB are 1296 (request) and 85 (response). For 1 KB Get
operation, these are 136 (request) and 1091 (response) bytes. It is to be noted that the commu-
nication time decreases considerably in the UCR-based HBase design. Communication
preparation time is also reduced in this design. In socket based design, the serialized object
has to be copied into Java socket buffer. But our design bypasses this, which avoids the
copy overhead.

![Graphs showing latency and throughput](image)

Figure 8.4: Single Server, Multiple Clients, HBase Get on Cluster A, 1 KB Message Size.

As we can observe from Figure 8.3, communication cost is one of the major factors
contributing to the overall operation latency. The total communication time for 1 KB Put
operation over UCR is 8.9 $\mu s$, where as it is 168.3 $\mu s$, 78.5 $\mu s$ and 57.1 $\mu s$ for 1 GigE, IPoIB
and 10 GigE networks, respectively. Our design achieves a performance improvement of
6 X times over 10 GigE (fastest among the socket versions). Similar trend is observed for
Get operation, as indicated in the figure.

Overall, UCR-based design decreases the communication time substantially and re-
duces the overall latency perceived by end user.
8.2.4 YCSB (Single server and Multiple clients)

In this experiment, we use the YCSB benchmark and measure the HBase Get operation latency. We use single region server - multiple clients configuration for this experiment. All the clients issue Get operation on a same record. We restrict the server working set to just one record, so that all the queries are serviced from the server memory, avoiding hard disk access anomalies.

Figure 8.4(a) shows the average latency for a Get operation for varying number of clients. As it can be observed from the figure, UCR-based design substantially reduces the operation latencies as compared to socket-based transports. For 8 clients, our design reduces the latency by around 26% as compared to 10 GigE network. As the number of clients increase, the operation latency rises due to heavier workload on the server. The performance improvement using UCR can be observed even with increased number of clients.

Figure 8.4(b) denotes the transaction throughput for the same experiment. The total throughput increases as more clients are added because of higher level of concurrency in request handling. It is to be noted that UCR based design achieves very good throughput scalability with increasing number of clients. For 16 clients, our design achieves a throughput of 53.4 K ops/sec, which is 27% higher than 10 GigE network.

8.2.5 YCSB: Multi-servers and Multi-clients

In this experiment we use four region server nodes and vary the number of client nodes from 1 to 16. We prepare client nodes in two configurations, 1 client thread per node and 8 client threads per node. For each of these configurations, we use different workloads: 1) 100% Get operations (Read Workload), 2) 100% Put operations (Write
(a) Read: Latency

(b) Read: Throughput

(c) Write: Latency

(d) Write: Throughput

Figure 8.5: Multi-Servers Multi-Clients, Read-only or Write-only Workload on Cluster A (1 thread/node, 1 KB message)

(a) Read Latency

(b) Write Latency

(c) Throughput

Figure 8.6: Multi-Servers Multi-Clients, Read-Write Workload on Cluster A (1 thread/node, 1 KB message)
Workload), and 3) 50% Get + 50% Put operations (Read-Write Workload). We disable caching at the client side by deleting the following statements in YCSB benchmark:

```java
_hTable.setAutoFlush(false); _hTable.setWriteBufferSize(1024 * 1024 * 12);```

This models the benchmark behavior to that of real world workloads. We use four HRegionServers in this experiment, with 320,000 1 KB records (320 MB data) evenly distributed among all four servers. Each HRegionServer hosts 80 MB data which is completely cached in memory. The client threads perform queries according to either Zipfian distribution or Uniform distribution. We do not find any perceivable difference in terms of performance between these two distributions. Therefore only the results from Zipfian distribution are reported. Also, because of limited number of 10 GigE adapters available, we have only 12 client nodes in the experiments with 10 GigE networks.

**Read Workload, One Thread per Client Node:** In this experiment, we vary the number of client nodes from 1 to 16, with each node running one YCSB thread. Each YCSB client issues Get operations to the HRegionServers. Similar to previous results, UCR-based design significantly reduces the query latency as indicated in Figure 8.5(a). For 8 clients, the operation latency using our design is 174.6 µs. This is 43% lower than the 10 GigE operation latency. As the latency is reduced, UCR-based design almost doubles the aggregated operation throughput as compared to the 10 GigE network.

**Write Workload, One Thread per Client Node:** This experiment is similar to the previous experiment. Each client node runs one YCSB thread that issues Put operations to the HRegionServers. As shown in Figure 8.5(c), significant performance improvement is observed for UCR-based design. For 16 clients, UCR design achieves a throughput of 41.7 K ops/sec, which is 20% higher than IPoIB network.
Read-Write Workload, One Thread per Client Node: In this workload each YCSB thread issues equal amounts of Get/Put operations. Among the three socket-based transports 10 GigE performs the best. The results are depicted in Figure 8.6. Compared to 10 GigE, UCR reduces the read and write latency by 24% and 15%, respectively for 8 clients. Our design also boosts up the overall transaction throughput by up to 23%.

Read Workload, Eight Threads per Client Node: In this experiment, each client node runs 8 YCSB threads to perform Get operations. We vary the number of client nodes from 1 to 16 so that the total number of client threads ranges from 8 to 128. Figure 8.7(a) depicts that all the socket-based transports perform almost analogously, with 10 GigE slightly better. The operation latency rises with increased workload from more client threads. More client threads also lead to higher throughput. Across all the ranges, UCR-based design always yields the lowest latency and highest throughput. For 64 clients, our design reduces the latency by around 26% as compared to 10 GigE network. Our design improves the operation latency by around 25% as compared to IPoIB for 128 clients.

Write Workload, Eight Threads per Client Node: In this workload, each client thread issues Put operations to the HBase HRegionServers. The total number of client threads are varied from 8 to 128. The latency and throughput results are given in Figure 8.7(c). The write operation latency is reduced by around 15% as compared to 10 GigE, for 64 clients. For 128 clients, the write throughput is boosted up by 22% over the throughput obtained using IPoIB network.

Read-Write Workload, Eight Threads per Client Node: In this workload illustrated in Figures 8.7, each client thread performs both Put and Get operations. We disable client side query batching in YCSB to force each client request directly go to HBase server without
being cached. We observe very high latency with 10 GigE and 1 GigE. For 64 clients, the throughput improvement is around $3 \times$ times over 1 GigE and 10 GigE networks.

![Graphs showing latency and throughput for different network configurations.](image)

Figure 8.7: Multi-Servers Multi-Clients, Read-only or Write-only Workload on Cluster A (8 threads/node, 1 KB message)
8.3 Summary

To summarize, we took on the challenge to improve HBase data query performance by improving the communication time. We performed detailed profiling to reveal the communication overheads caused by Java sockets, and showed that communication played an important role in the overall operational latency. We identified this as an opportunity to take advantage of high performance networks to improve HBase operation performance.

We designed a hybrid hierarchical communication mechanism to incorporate both conventional sockets and high performance InfiniBand. By leveraging the high-throughput and low-latency InfiniBand network, we were able to substantially drive down HBase data query latency and boost up transaction throughput. We exploited the InfiniBand RDMA capability and adopted it to match with the object delivery model used by HBase. Our design achieves a latency of as low as 43.7 $\mu$s for a 1KB payload Get operation, which is around $3.5X$ times better than 10 GigE sockets. Our design also substantially increases the operation throughput. In YCSB 50%-read-50%-write experiment with 64 clients, our design delivers a throughput $3X$ times higher than 10 GigE.

Figure 8.8: Multi-Servers Multi-Clients, Read-Write Workload on Cluster A (8 threads/node, 1 KB message)
Chapter 9: Impact on the HPC and Big Data Communities

9.1 Impact on the Design and Use of MPI and PGAS Libraries

MVAPICH2-X, proposed as part of this work, has changed the way MPI and PGAS libraries are designed. It introduced the concept of hybrid programming supported with a ‘single unified’ runtime to the HPC community. Many other MPI implementations like Open MPI [62] have followed suite to support MPI and OpenSHMEM Communication with a single runtime, and UPC stacks such as Berkeley UPC [48] have employed optimization techniques similar to the ones proposed in this report. This work has also influenced the way applications with mixed regular and irregular characteristics are being designed. Application developers can now mix and match MPI and PGAS models based on the communication characteristics without any performance or resource penalty. This reduces the complexity of communication designs in applications, and improves programmability. The advantages of hybrid MPI+PGAS model has also influenced the MPI standards, and many PGAS features are proposed in MPI 3.0 [56].

9.2 Software Release and Wide Acceptance

MVAPICH2 [57, 65], is an open-source MPI implementation over modern high-speed networks such as InfiniBand, 10GigE/iWARP and RDMA over Converged Ethernet (RoCE).
MVAPICH2 delivers the best performance, scalability and fault tolerance for high-end computing systems and servers using InfiniBand, 10GigE/iWARP and RoCE networking technologies. This software is being used by more than 2,150 organizations world-wide in 72 countries and is powering some of the top supercomputing centers in the world, including the 7th ranked TACC Stampede, 13th ranked Tsubame 2.5, and 23rd ranked Pleaides (based on June 2014 Top500 Rankings). As of July ’14, more than 219,000 downloads have taken place from this project’s site. This software is also being distributed by many InfiniBand, 10GigE/iWARP and RoCE vendors in their software distributions.

The proposed thesis is released as MVAPICH2-X software package (released along with MVAPICH2 software package), which provides support for hybrid MPI+PGAS (UPC, OpenSHMEM) programming models with unified communication runtime for emerging exascale systems. MVAPICH2-X enables application scientists to redesign applications using hybrid programming models, without having to worry about programming model intricacies. MVAPICH2-X is the first ever runtime to support both MPI and PGAS programming models. The first release of MVAPICH2-X was on September 2012, and by July 2014 there have been nearly 3,600 downloads, world-wide. They are deployed and widely used on large scale Clusters including Stampede. Special mailing lists are kept for MVAPICH2-X release and announcements and for discussing MVAPICH2-X usage discussions. On top of this, comprehensive user-guide is maintained from project website.

The RDMA-based design of Memcached will also be released from project site. UCR is also being used as the communications substrate in other active research projects in Hadoop-RDMA [30], such as high performance design of MapReuce and HDFS. UCR has also been used in industry collaboration projects (with IBM, Mellanox) for research in improving communication performance of data-center middleware.
Chapter 10: Future Research Directions

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

10.1 Hierarchical Hybrid MPI+PGAS Models

Many-core processors are expected to be an integral part of system designs targeted for exascale computing. It is expected that future architectures will focus on hosted many-core designs, thereby avoiding PCIe bottlenecks, that limit communication performance on current generation many-core clusters. Such architectures will have some heavy cores and a large number of tiny cores on a single chip (like AMD APUs and NVIDIA Tegra mobile processors [61]), giving rise to heterogeneous many-core processors. ARM-based processors are also expected to be part of such solutions [16]. Another alternative is a chip with large number of homogeneous, light-weight general purpose cores (e.g., Knights Landing [34]). Though, such an architecture appears to reduce the complexity of communication, it brings other significant challenges including low compute power, very limited memory per core and slower sequential processing speed. In summary, these DMN architectures lead to the following fundamental characteristics for next generation HEC systems: 1) Extreme parallelism with many cores, 2) Heterogeneity in cores, 3) Low memory per core, and 4) Hierarchical networks.
In order to extract the maximum performance out of these emerging architectures, programming models shall match the architecture characteristics. Thus the traditional ‘flat’ models might not be the best way to program such systems. Through the Unified Communication Runtime, designed as part of this thesis, hierarchical MPI+PGAS programming models are possible. Even though simple funneled, nested funneled models are proposed in earlier studies, these did not consider the hierarchy available in the system [24]. One possible future research direction is to explore such models, and redesign applications on these upcoming architectures.

10.2 Exploring Streaming Semantics in UCR

Data streams arise in many application domains, such as sensor processing, network monitoring and financial analysis. For such applications, low-latency and high bandwidth are critical. However, many existing big-data middleware and even HPC applications rely on traditional BSD sockets for the streaming semantics. In this context, it will be beneficial for the community to have high performance streaming interfaces available as part of the programming model or the underlying middleware. One possible solution is to support high performance streaming communication as part of UCR. This allows streaming communication interfaces to be easily implemented for MPI, PGAS, and Big data middleware.

10.3 Impact of Designs on Energy-constrained Environments

Energy-constrained environments will become increasingly common as the HPC community heads towards exascale computing. Heterogeneous configurations with massively parallel accelerators and co-processors coupled with low-power host processors (e.g. ARM) are being explored as a possible alternative for building such systems. For applications to
efficiently run on such clusters, computation kernels ought to be designed in a highly parallel fashion to take advantage of the accelerators or co-processors while the host processors are only involved in scheduling/progressing computation and communication phases. The work done as part of this thesis demonstrates how one-sided communication semantics backed by RDMA-based runtime-level designs can significantly reduce synchronization overheads and can completely hide communication overheads through overlap. The need for such designs will become imperative as we move into energy-constrained environments with slower host processors described above. One-sided, offloaded communication will reduce the overhead on the low-power processor preventing it from becoming the bottleneck in the system. As these architectures mature, research to evaluate the impact of these designs on energy-footprint of applications will be essential. New runtime-level techniques and programming model-level extensions may have to be devised based on the requirements and capabilities of these new architectures.
Chapter 11: Conclusion and Contributions

MPI has been the prevailing communication middleware in HPC, for more than two decades. Even though it has been successful in developing regular, iterative applications, it can be very difficult to use MPI and maintain performance for irregular, data-driven applications. PGAS programming model presents an attractive alternative for designing such applications and provides higher productivity. It is widely believed that parts of applications can be redesigned using PGAS models - leading to hybrid MPI+PGAS applications. In order to fully leverage the performance benefits offered by the modern HPC systems, a unified communication runtime that offers the advantages of both MPI and PGAS programming models is critical.

In this dissertation, we designed and developed “MVAPICH2-X” - a high performance and scalable ‘Unified Communication Runtime’ which supports both MPI and PGAS programming models. Experimental evaluations reveal the MVAPICH2-X runtime performs better for pure UPC, pure OpenSHMEM, and hybrid MPI+PGAS applications. For UPC NAS benchmarks CG and MG (class B) at 128 processes, MVAPICH2-X design outperforms the GASNet-IBV runtime by 10% and 23%, respectively. Further, memory scalability analysis reveals that our design is highly scalable. OpenSHMEM microbenchmark evaluations indicated 41% lower atomic operation latency, 33% and 40% improved performance for collective routines Collection and Reduction for 512 processes and 30% better
performance for Barrier operation with 512 processes. Further, OpenSHMEM application evaluations also exhibit similar performance results. 2D Heat Modeling application execution time with 512 PE’s is improved by 45%.

We also proposed a novel high performance design using MVAPICH2-X for OpenSHMEM on Intel MIC architecture. We proposed extensions for allowing non-uniform memory allocations, which are suitable for upcoming heterogeneous memory architectures. We presented proxy-based runtime designs for providing low latency and high bandwidth for communication operations, by efficiently by-passing high overhead communication paths. Our benchmark evaluations reveal 4X-6X reduction in operation latencies. Application level evaluations indicate 28% improvement over the default designs. Further, the application re-designs using the proposed design provides magnitudes of performance improvement compared to the default symmetric mode.

To further evaluate and strengthen our designs for unified runtime, we re-designed pure MPI versions of Graph500 and Out-of-Core Sort using hybrid MPI+PGAS. We presented a detailed analysis of existing MPI based Graph500 implementation and exposed critical bottlenecks and presented a scalable and high performance design using MPI and OpenSHMEM constructs. Performance evaluations using MVAPICH2-X Unified Runtime show a reduction in Graph500 traversal time by 59%, compared to the best performing MPI design at 8,192 cores. Scalability analysis indicates that the hybrid design demonstrates good strong and weak scaling characteristics. At this scale, hybrid design performs 8X better than the MPI design which has the same communication pattern and volume. Our evaluations with a unified runtime and with separate runtimes highlight the need for a unified communication runtime for hybrid programming models. Performance evaluations of
Hybrid MPI+PGAS Out-of-Core Sort revealed that the hybrid design improves the performance significantly. At 4,096 processes, the sort execution time is reduced by 51% using the hybrid design as compared to original MPI based design. Our scalability experiments indicate that hybrid design demonstrates good strong and weak scaling characteristics.

For Big Data middleware, we presented a light-weight, high performance and highly scalable communication runtime design. This design abstracts out the intricacies of complex interfaces of high performance network APIs and provides simple interfaces consisting of active messages and one-sided operations. In this dissertation, we redesigned two popular Big Data middleware — Memcached, and HBase — using the Unified Communication Runtime. Memcached performance evaluations indicated that latency of Memcached Get of 4KB size can be brought down to 12 $\mu$s using ConnectX InfiniBand QDR adapters. Latency of the same operation using older generation DDR adapters is about 20 $\mu$s. These numbers are about a factor of four better than the performance using 10 GigE with TCP Offload. In addition, these latencies of Get requests over a range of message sizes are better by a factor of five to ten compared to IP over InfiniBand and Sockets Direct Protocol over InfiniBand. Further, throughput of small Get operations can be improved by a factor of six when compared to Sockets over 10 Gigabit Ethernet network. The HBase design achieved a latency of as low as 43.7 $\mu$s for a 1KB payload Get operation, which is around $3.5X$ times better than 10 GigE sockets. Our design also substantially increased the operation throughput. In YCSB 50%-read-50%-write experiment with 64 clients, our design delivered a throughput $3X$ times higher than 10 GigE.
Bibliography


223


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


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


[65] D. K. Panda, K. Tomko, K. Schulz, and A. Majumdar. The MVAPICH Project: Evolution and Sustainability of an Open Source Production Quality MPI library for HPC. In *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Int’l Conference on Supercomputing (WSSPE)*.


