Designing Efficient MPI and UPC Runtime for Multicore Clusters with InfiniBand, Accelerators and Co-Processors

Dissertation

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

By

Miao Luo, M. Sc.

Graduate Program in Computer Science and Engineering

The Ohio State University

2013

Dissertation Committee:

Prof. D. K. Panda, Advisor
Prof. P. Sadayappan
Prof. R. Teodorescu
Abstract

High End Computing (HEC) has been growing dramatically over the past decades. The emerging multi-core systems, heterogeneous architectures and interconnects introduce various challenges and opportunities to improve the performance of communication middlewares and applications. The increasing number of processor cores and Co-Processors results in not only heavy contention on communication resources, but also much more complicated levels of communication patterns.

Message Passing Interface (MPI) is the dominant parallel programming language for HPC application area in the past two decades. MPI has been very successful in implementing regular, iterative parallel algorithms with well defined communication pattern. Instead, the Partitioned Global Address Space (PGAS) programming model provides a flexible way for these applications to express parallelism. Different variations and combinations of these programming languages present new challenges in designing optimized programming model runtimes, in terms of efficient sharing of networking resources and efficient work-stealing techniques for computation load balancing across threads/processes, etc.

Middlewares play a key role in delivering the benefits of new hardware techniques to support the new requirement from applications and programming models. This dissertation aims to study several critical contention problems of existing runtimes, which supports popular parallel programming models (MPI and UPC) on emerging multi-core/many-core
systems. We start with shared memory contention problem within existing MPI runtime. Then we explore the network throughput congestion issue at node level, based on Unified Parallel C (UPC) runtime. We propose and implement lock-free multi-threaded runtimes for MPI/OpenMP and UPC with multi-endpoint support, respectively. Based on the multi-endpoint design, we further explore how to enhance MPI/OpenMP applications with transparent support for collective operations and minimal modifications for point-to-point operations. Finally we extend our multi-endpoint research to include GPU and MIC architecture for UPC and explore the performance features.

Software developed as a part of this dissertation is available in MVAPICH2 and MVAPI CH2-X. MVAPICH2 is a popular open-source implementation of MPI over InfiniBand and is used by hundreds of top computing sites all around the world. MVAPICH2-X supports both MPI and UPC hybrid programming models on InfiniBand clusters and is based on MVAPICH2 stack.
Dedicated to my parents, Caiqin and Zhaoquan.
Acknowledgments

It’s an invaluable experience to spend my PhD study with so many great people in NOWLAB.

First of all, I would like to thank my adviser and mentor, Prof. D. K. Panda, for his guidance and support during my 6-year PhD study. I am sincerely thankful for his efforts and help in advising my research and leading me to the computer science research community. I have been learning a lot from his hard work and dedication as an academic researcher.

I would like to thank my committee members Prof. Sadayappan and Prof. Teodorescu for their valuable guidance and suggestions.

I’m very lucky to have spent my PhD life working closely with many talented students and researchers at NOWLAB. I would like to thank the senior colleagues in NOWLAB including Dr. Matthew Koop, Dr. Sayantan Sur, Dr. Hao Wang, Dr. Jerome Vienne, Dr. Xiangyong Ouyang, Dr. Khaled Hamidouc, and Dr. Xiaoyi Lu. I’m also grateful to have worked with many colleagues in NOWLAB including Hari, Krishna, Sreeram, Raghu, Jithin, Mingzhe, Akshay, Nusrat, Wasi, Devendar, Joanthan, and Mark. I thank them all for their innumerable discussions on technical and non-technical issues, as well as their friendship.

I would like to thank my parents. Their selfless love is always the endless motive power for me.
Vita

April 19, 1985 .......................... Born - Hengshui, China

September 2003 - July 2007 ............... Bachelor in Electronic and Information Engineering,
                                          Beijing University of Posts and Telecommunications, China

September 2008 - February 2009 .......... Graduate Teaching Associate,
                                          The Ohio State University.

June 2011 - September 2011 ............... Research Intern,
                                          Lawrence Berkeley National Laboratory,
                                          California.

March 2009 - Present ..................... Graduate Research Associate,
                                          The Ohio State University.

Publications


Fields of Study

Major Field: Computer Science and Engineering
Studies in:

Computer Architecture  Prof. D. K. Panda
Computer Networks     Prof. D. Xuan
Software Systems       Prof. S. Parthasarathy
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xv</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xix</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Message Passing Interface (MPI)</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 MVAPICH2 Library</td>
<td>6</td>
</tr>
<tr>
<td>2.2 PGAS Programming Models</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Unified Parallel C</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Multi-core and Heterogeneous Architectures</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1 GPGPU Clusters</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 MIC Architecture</td>
<td>10</td>
</tr>
<tr>
<td>2.4 InfiniBand Architecture</td>
<td>11</td>
</tr>
<tr>
<td>3. Problem Statement</td>
<td>13</td>
</tr>
</tbody>
</table>

ix
4. Improve MPI Inter-Process Communication over Shared Memory
   4.1 Scalability Issues in Existing Designs
   4.1.1 Frequent Shared Structure Access
   4.1.2 High Latency in Fair Polling
   4.1.3 Memory Usage for Large Memory Transfer
      4.1.3.1 Contentions accessing shared send buffer pool and shared receive queue
      4.1.3.2 Possible Waste of Shared Memory Buffer Pool
   4.2 Designs
      4.2.1 Shared Tail Cyclic Buffer
      4.2.2 State-Driven Polling Scheme
      4.2.3 On-demand Global Shared Memory Pool - memory efficient large message transfer
   4.3 Discussion about Memory Usage
   4.4 Performance Evaluation
      4.4.1 Micro-benchmark Evaluation
      4.4.2 Application Evaluation
         4.4.2.1 Graph500
         4.4.2.2 NAS Benchmark
   4.5 Related Works
   4.6 Summary

5. Congestion Avoidance on Multi-core High Performance Computing Systems
   5.1 Network Performance Characterization
   5.1.1 Experimental Setup
   5.1.2 Analysis
   5.2 Core Stateless Congestion Avoidance
      5.2.1 Admission Control Policies
      5.2.2 Admission Control Implementations
   5.3 Evaluations
      5.3.1 Microbenchmark Evaluation
      5.3.2 Building a Congestion Avoidance Policer
      5.3.3 All-To-All Performance
      5.3.4 Application Benchmarks
   5.4 Discussion
   5.5 Related Work
   5.6 Summary
6. Reducing Network Contention of Multi-threaded UPC Runtime through Multi-endpoint ......................................................... 69
   6.1 Analysis of Design Alternatives for UPC Runtimes .................. 70
   6.2 Design Details of Multi-threaded Runtime ............................ 75
   6.3 Load Balancing for Irregular Applications Based on Multi-endpoint ... 77
       6.3.1 Dedicated Communication Threads ............................. 78
       6.3.2 Work Stealing for Efficient Asynchronous Remote Methods ... 78
   6.4 Performance Evaluation ............................................... 79
       6.4.1 Experimental Platform ......................................... 80
       6.4.2 Microbenchmark Level Evaluation ............................... 80
       6.4.3 Message Rate Evaluation ....................................... 83
       6.4.4 Dedicated Communication Thread - Bandwidth Evaluation ... 85
       6.4.5 Work Stealing - Communication Evaluation ................... 86
       6.4.6 Work Stealing - Computation Evaluation ...................... 88
   6.5 Related Work ....................................................... 90
   6.6 Summary ........................................................... 92

7. High Performance Lock-free Multi-endpoint Runtime for MPI/OpenMP Hybrid Applications ......................................................... 93
   7.1 Motivation .......................................................... 94
   7.2 Designing Multi-endpoint Communication Runtime .................. 98
       7.2.1 Lock-free Communication Runtime Based on Multi-endpoint ... 98
       7.2.2 Multi-endpoint Impact on Network Performance ................. 100
   7.3 Multi-Endpoint support for Point-to-Point operations ............. 102
   7.4 Transparent Re-design for Collectives Operation .................... 104
       7.4.1 Alltoallv ..................................................... 104
       7.4.2 Allgatherv .................................................... 107
   7.5 Experimental Results ............................................... 110
       7.5.1 Experimental Setup ........................................... 110
       7.5.2 Multi-endpoint Runtime Threads Latency ....................... 112
       7.5.3 Point-to-Point Micro-benchmark ............................... 112
       7.5.4 Collective Micro-benchmark .................................... 113
       7.5.5 Application Evaluations ....................................... 114
           7.5.5.1 NAS Benchmark ......................................... 114
           7.5.5.2 P3DFFT Kernel Lib .................................... 115
   7.6 Discussion ......................................................... 116
   7.7 Related Work ...................................................... 118
   7.8 Summary .......................................................... 119
8. Improving GPU to GPU Communication for UPC Runtime over InfiniBand . . 120

8.1 Motivation ................................................................. 121
8.2 Design ................................................................. 123
  8.2.1 GPU Global Address Space for Multi-threaded Runtime ............ 123
  8.2.2 Design for Remote Memory Operation .......................... 124
  8.2.3 Helper Thread for Improved Asynchronous Access ............... 127
8.3 Evaluations .............................................................. 130
  8.3.1 Experimental Platforms ........................................ 130
  8.3.2 Micro-benchmark Evaluation .................................... 130
  8.3.3 Helper Thread Evaluation ....................................... 133
  8.3.4 Sample Application Evaluation ................................. 134
8.4 Related Work .......................................................... 135
8.5 Summary ............................................................... 136

9. UPC on MIC: Early Experiences with Native and Symmetric Modes . . . . 138

9.1 Motivation ................................................................. 139
9.2 Alternatives for running UPC on MIC ................................. 141
  9.2.1 UPC Runtime on MIC: Process-based or Thread-based? ........ 141
  9.2.2 Remote Memory Access between MIC and HOST .................. 144
9.3 Evaluations .............................................................. 145
  9.3.1 Experimental Platforms ........................................ 145
  9.3.2 Micro-benchmark evaluations .................................... 146
    9.3.2.1 Native Mode Point-to-Point Evaluations ............... 146
    9.3.2.2 Native Mode Collective Evaluations .................... 149
    9.3.2.3 Symmetric Mode Point-to-Point Evaluations .......... 149
  9.3.3 NAS Benchmark ................................................... 150
  9.3.4 Random Access .................................................. 152
  9.3.5 UTS Benchmark .................................................. 152
9.4 Discussion ............................................................. 153
9.5 Related Work .......................................................... 154
9.6 Summary ............................................................... 155

10. Software Distribution .................................................. 156

11. Conclusions and Future Research Directions .............................. 158

  11.1 Summary of Research Contributions ................................ 158
    11.1.1 Improve MPI Inter-Process Communication over Shared Memory 158
11.1.2 Congestion Avoidance on Multi-core High Performance Computing Systems

11.1.3 Reducing Network Contention of Multi-threaded UPC Runtime through Multi-endpoint

11.1.4 High Performance Lock-free Multi-endpoint Runtime for MPI OpenMP Hybrid Applications

11.1.5 Improving GPU to GPU Communication for UPC Runtime over InfiniBand

11.1.6 UPC on MIC: Early Experiences with Native and Symmetric Modes

11.2 Future Research Directions

Bibliography
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Number of Bus Operations Over Shared Control Structure Per Communication</td>
</tr>
<tr>
<td>4.2</td>
<td>CG (CLASS D) Performance on Trestles (Time: sec)</td>
</tr>
<tr>
<td>7.1</td>
<td>P3DFFT Performance on Stampede (Time: sec)</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Problem Space of this Dissertation</td>
</tr>
<tr>
<td>4.1</td>
<td>Pseudocode for Shared Memory Communication in Existing Designs</td>
</tr>
<tr>
<td>4.2</td>
<td>Contentions from accessing shared control structure</td>
</tr>
<tr>
<td>4.3</td>
<td>Shared Tail Cyclic Buffer Structure</td>
</tr>
<tr>
<td>4.4</td>
<td>Receiver Polling State Machine</td>
</tr>
<tr>
<td>4.5</td>
<td>Dynamic Queue Pin-down from Global Share Memory Pool</td>
</tr>
<tr>
<td>4.6</td>
<td>Micro-benchmark Evaluations: One Pair Ping-Pong Latency</td>
</tr>
<tr>
<td>4.7</td>
<td>Micro-benchmark Evaluations: Multi-Pair Message Rate</td>
</tr>
<tr>
<td>4.8</td>
<td>Micro-benchmark Evaluations: One Pair Bandwidth</td>
</tr>
<tr>
<td>4.9</td>
<td>Application Evaluation on 64-core AMD Bulldozer System</td>
</tr>
<tr>
<td>4.10</td>
<td>Communication Topology: NAS-CG application with 512 procs and class D</td>
</tr>
<tr>
<td>5.1</td>
<td>Concurrency Congestion: node throughput drops when multiple cores are active at the same time. We assume each core has one message to send and we plot the speedup of using cores/x rounds of communication over all cores active.</td>
</tr>
<tr>
<td>5.2</td>
<td>Rate Congestion: node throughput drops when cores have multiple outstanding messages. We assume each core has to send 1,024 messages and we plot the speedup of using 1,024/x rounds of communication with x = (1,2,4..) messages over sending 1,024 messages at the same time.</td>
</tr>
</tbody>
</table>
5.3 Concurrency and Rate Congestion in MPI. Note that MPI (two-sided) exhibits even larger performance degradation, up to 600%, than UPC (one-sided). ................................................................. 46

5.4 Software architecture of the Proxy implementation. An admission control policy layer can be easily added behind the servers. ............................. 51

5.5 Performance impact of Rate (IRT), Token (ITT) and Proxy congestion avoidance on InfiniBand. We plot the speedup of applying our congestion avoidance while increasing the number of outstanding messages per core (1, 2, 4, 8, 16). ................................................................. 53

5.6 Performance impact of Token (ITT) and Proxy congestion avoidance on Gemini. We allow an increasing number of outstanding messages per core and plot the speedup of applying our congestion avoidance mechanisms. . 53

5.7 Overhead of Congestion Avoidance with Active Cores. We plot the speedup for an increasing number of active cores, number of messages per core and message sizes. We vary the message size from 8B to 512KB in increasing powers of 2 and there is one data point per 8, 16, ..., 512KB. ................................. 54

5.8 Impact of IRT, ITT and Proxy congestion avoidance on all-to-all performance. We plot the speedup of MPI and that of single implementation running on top of congestion avoidance (IRT, ITT, Proxy). The performance baseline is an overlapped implementation in UPC. ................................. 58

5.9 RandomAccess on 1,024 cores InfiniBand. We plot the speedup relative to a baseline implementation using blocking communication. ................................. 60

5.10 The NAS Parallel Benchmarks on InfiniBand. We plot the speedup relative to a baseline implementation using blocking communication. ................................. 60

6.1 Non Multi-threaded and Multi-threaded choices for UPC Runtime .......... 71

6.2 Multi-Dimensional Comparison of Design Alternatives: Dotted line indicates coverage of design space by multi-threaded runtime with multiple network endpoints ......................................................... 74

6.3 UCR: Thread Mapping with Multi-Endpoint ................................. 76
6.4 Small message latency performance ........................................81
6.5 Large message latency performance .....................................81
6.6 Bandwidth performance ....................................................81
6.7 Message Rate Performance ................................................83
6.8 Dedicated Communication Thread Design - Bandwidth Evaluation ....87
6.9 Work-Stealing Evaluation with unbalanced computation requests ....87
7.1 Point-to-Point communication characteristics with varying number of cores per node .......................................................... 96
7.2 Overview of the Multi-Endpoint framework ..........................97
7.3 Components of the multi-endpoint based lock-free communication runtime 98
7.4 Comparison of the Communication Overlapping in Process-based Runtime and Multi-endpoint Runtime ............................................ 101
7.5 Minimal modifications to expose multi-endpoint for point-to-point operations 103
7.6 Alltoallv Algorithm based on Multi-Endpoint ....................... 106
7.7 Allgatherv Algorithm based on Multi-Endpoint ......................107
7.8 Multi-Pair latency performance across different runtimes ..........110
7.9 Micro-benchmark Evaluations for Point-to-Point Optimization ........111
7.10 Micro-benchmark Evaluations: Alltoallv ..............................111
7.11 Micro-benchmark Evaluations: Allgatherv ............................111
7.12 NAS Benchmark on Stampede ..........................................115
7.13 Increased memory footprint per endpoint on Stampede ..........115
8.1 UPC and CUDA Operations for Manual Remote Device Memory Access 121
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2</td>
<td>Pseudocode for Shared Memory Allocation with Extended APIs</td>
</tr>
<tr>
<td>8.3</td>
<td>UPC/CUDA Remote Device Memory Operation through RDMA Fastpath Design</td>
</tr>
<tr>
<td>8.4</td>
<td>UPC/CUDA Remote Device Memory Operation for Large Message</td>
</tr>
<tr>
<td>8.5</td>
<td>Dedicated Helper Thread Carry out Device Memory Access for Busy UPC Thread</td>
</tr>
<tr>
<td>8.6</td>
<td>Micro-benchmark Evaluations: memput latency</td>
</tr>
<tr>
<td>8.7</td>
<td>Micro-benchmark Evaluations: memget latency</td>
</tr>
<tr>
<td>8.8</td>
<td>Helper Thread Evaluation with memput/memget</td>
</tr>
<tr>
<td>8.9</td>
<td>UPC/CUDA Matrix Multiplication</td>
</tr>
<tr>
<td>9.1</td>
<td>UPC on MIC: native and symmetric modes</td>
</tr>
<tr>
<td>9.2</td>
<td>Leader-to-all Connection Mode</td>
</tr>
<tr>
<td>9.3</td>
<td>Intra-MIC Evaluations: single pair memput latency</td>
</tr>
<tr>
<td>9.4</td>
<td>Intra-MIC Evaluations: multi-pair memput latency</td>
</tr>
<tr>
<td>9.5</td>
<td>Intra-MIC Evaluations: collective latency</td>
</tr>
<tr>
<td>9.6</td>
<td>Host-to-MIC Evaluations: multi-pair memput latency</td>
</tr>
<tr>
<td>9.7</td>
<td>MIC-to-HOST Evaluations: multi-pair memput latency</td>
</tr>
<tr>
<td>9.8</td>
<td>NAS benchmark performance with native mode</td>
</tr>
<tr>
<td>9.9</td>
<td>RandomAccess and UTS benchmark performance with native mode</td>
</tr>
</tbody>
</table>
List of Algorithms

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Pseudocode for Congestion Avoidance</td>
<td>56</td>
</tr>
<tr>
<td>7.1 Group Pair-wise Algorithm for Alltoallv based on Multi-Endpoint Design</td>
<td>105</td>
</tr>
<tr>
<td>7.2 Double-Ring Algorithm for Allgatherv based on Multi-Endpoint (Part 1)</td>
<td>108</td>
</tr>
<tr>
<td>7.3 Double-Ring Algorithm for Allgatherv based on Multi-Endpoint (Part 2)</td>
<td>109</td>
</tr>
</tbody>
</table>
Chapter 1: INTRODUCTION

High End Computing (HEC) has been growing significantly during past decades. The HEC systems allow scientists and engineers to tackle grand challenges in their research areas. The deployment of modern and future ultra-scale systems is fueled by the increasing utilization of multi-core and many-core environments. We observe that the multi-core systems are increasingly widespread in modern commodity super computing clusters, due to the advent of multi-core processing. Systems with 64 cores per node are being deployed, while systems with 128 or even larger number of cores per node are being developed. Recently, the emerging heterogeneous architecture with General-purpose GPU (GPGPU) and Co-Processors are getting more attentions for their remarkable peak performance and cost efficiency of the devices. The GPU devices and Co-Processors are packed inside one single node to accelerate computation. High performance interconnects, such as InfiniBand, are also being increasingly deployed for the configuration of commodity clusters at a relatively modest cost. Based on the June 2013 TOP500 [19] ranking, there are more than 11% systems using accelerator/co-processor technology, and more than 41% systems using InfiniBand interconnects.

MPI is the dominant parallel programming language for HPC application area in the past two decades. MPI has been very successful in implementing regular, iterative parallel algorithms with well defined communication pattern. However, it can be very difficult
to use MPI and maintain performance for irregular, data-driven applications, especially on heterogeneous architectures. Instead, the Partitioned Global Address Space (PGAS) programming models provides a flexible way for these applications to express parallelism. In the meanwhile, GPGPUs introduce different programming models, such as CUDA [87], OpenCL [13] and OpenACC [12], for a massive parallelism on accelerators.

The emerging multi-core system, heterogeneous architecture and interconnects introduce various challenges and opportunities to improve the performance of communication middlewares and applications. The increasing number of processor cores and Co-Processors results in not only heavy contention on communication resources, but also much more complicated levels of communication patterns. On the Non Uniform Memory Access (NUMA) architecture, distance varies between network interface card, CPU cores, Co-Processors, accelerators and their separate memory banks. Due to the increasing portion of intra-node communication, shared memory channel has been emphasized for process-based middlewares, such as current MPI runtime libraries. However, since the existing shared memory based communication designs are mainly based on relatively small number of cores/node, multiple contention issues have become blockers towards delivering scalable and efficient performance in existing middleware runtimes. On the other hand, threads-based runtimes are being emphasized again due to its low-latency. Furthermore, threads-based runtime can provide low-level load balancing schemes by utilizing the feature of threads: shared address space. MPI+OpenMP mode has been utilized in many HPC applications. There is also a discussion of process-based or thread-based implementation for UPC runtime, due to the bad inter-node communication performance with thread-based runtime. All these variations present new challenges in designing optimized programming model runtimes in terms of efficient sharing of connections and networking resources cross
threads/processes and efficient work-stealing techniques for load balancing of computation across threads/processes, etc.

Clearly, middlewares play a key role in delivering the benefits of new hardware techniques to support the new requirement from applications. In this dissertation, we aim to study several critical contention problems of existing runtimes, which supports popular parallel programming models (MPI and UPC) on emerging multi-core/many-core systems. We start with shared memory contention problem within existing MPI runtime. Then we explore the network throughput congestion issue at node level, based on UPC runtime. We also provide a comparison of design alternatives of multi-threaded UPC runtime and utilize multi-endpoint to solve the network resource contention problem. We also design and implement lock-free multi-endpoint runtime to enhance the performance of MPI/OpenMP applications with transparent support for collective operations and minimal modifications for point-to-point operations. We extend our multi-endpoint research to include GPU and MIC architecture for UPC and discover the disadvantages of current programming models and implementations. Specifically we address the following questions:

1. Can we reduce the contention on shared memory resources of a modern multi-core system between processes, so that the performance and scalability of inter-process communication within a single compute node can be improved?

2. Can we identify the network throughput congestion problem of multi-core systems and provide core stateless mechanisms in runtime to control and improve such congestion at node level?
3. For multi-threaded runtime, such as multi-threaded UPC runtime, how to utilize the multi-endpoint designs to eliminate network access contentions, and furthermore, provide load balancing options at runtime level, for irregular applications?

4. As heterogeneous systems, such as accelerators and MIC architectures, are getting more and more popular, new resource contention problems are showing up between regular node and GPGPU or MIC. How should the middleware adopt new techniques in order to support different programming models and utilize the hardware resources in a more sufficient way?

5. Can we further optimize the multi-endpoint runtime designs in order to achieve better support for hybrid programming models?

The rest of this dissertation is organized as follows. Chapter 2 introduces a brief background and existing technologies related to MPI/PGAS language models, InfiniBand interconnects, and GPGPU/MIC architectures. In Chapter 3, we present the problem statement of this dissertation. Chapters 4-9 discuss the detailed approaches and results for these problems. Software distributions as part of this dissertation are described in Chapter 10. Chapter 11 provides the conclusion and possible future research directions.
Chapter 2: BACKGROUND

In this chapter, we discuss several important technologies that are related to our research. We first provide an overview of MPI and PGAS programming models. Then we discuss the emerging architecture trends in HPC area, including multi-core, GPGPU, and MIC systems. Finally we introduce InfiniBand architecture. This will provide appropriate background information for this dissertation.

2.1 Message Passing Interface (MPI)

Message Passing Interface (MPI) is a message-passing library interface specification. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. It has been a dominant parallel programming model in HPC area for developing parallel applications over the past two decades. Besides the “classic” message-passing model, extensions to the specification include collective operations, remote-memory access operations, dynamic process creation, and parallel I/O. The first draft proposal of MPI standardization, known as MPI-1, was put forward in 1992. Since then, MPI Forum [8] keeps working on corrections and extensions for the standard documents. MPI-3.0 is the latest MPI standard, which is just released September 2012, with extensions of collective operations and one-sided operations and a new Fortran 2008
binding. The recent widely adapted version of MPI Standard is MPI-2.2, which is sup-
ported by most existing open source implementations (Open-MPI [11], MPICH2 [9] and
MVAPICH2 [10]) and vendor implementations (Intel MPI [6], Cray MPT [45]).

2.1.1 MVAPICH2 Library

MVAPICH2 is an open-source MPI-2 implementation (conforming to MPI 2.2 stan-
dard and initial support for MPI-3) over InfiniBand [3], 10GigE/iWARP and RoCE [15].
Through underlying transport interfaces, MVAPICH2 supports multiple high performance
interconnects, including all Mellanox InfiniBand adapters, IBM eHCA adapters, Qlogic
adapters, Chelsio T3 adapters, Mellanox ConnectX-EN adapters with 10/40GigE switches
and so on. For intra-node communication, MVAPICH2 provides supports to Shared Mem-
ory Channel, LiMIC2 [66] and the Nemesis sub-channel derived from MPICH2 [48].

The Shared Memory Channel in MVAPICH2 uses pair-wise shared buffer for small
message transfer [37]. Each pair of sender and receiver keeps a control structure to record
the data transfer between them. By updating and reading these control structures, senders
and receivers can get the information about when and where to copy the data in or out of
the shared buffer. For large messages, MVAPICH2 utilizes shared sender buffer instead of
pair-wise scheme, in order to avoid large memory usage. Every sender keeps a large size
of shared memory as a send buffer pool separately. The buffer pool consists of small fixed-
size elements. After exchange hand-shaking messages with the receiver, the sender breaks
a large message and copies it into the elements from the shared send buffer pool. Then
the sender sends a notification message to the receiver. After detecting the notification, the
receiver can copy the data out from corresponding elements and mark the elements as used.
It’s the responsibility of senders to re-collect these used elements back to the shared buffer pool.

In Nemesis sub-channel, every sender keeps a send buffer pool that consists of multiple free cells in a queue manner. Each receiver keeps a shared receive queue that can be enqueued by all other processes on the same node. The sender first gets a free cell in its send buffer pool and fulfills the cell with data. Then the cell is enqueued into the receiver’s receive queue. The receiver polls its receive queue to get the pointer to the new data. Then the receiver copies data out and return the cell to the buffer pool of the sender. Multiple senders can insert cells to the same receive queue simultaneously, while multiple receivers can return cells to the same buffer pool at the same time. Thus the enqueue operations are atomic operations. To improve small message latency, Nemesis also keeps a pair-wise shared buffer so called “fastbox”. However, the “fastbox” can only hold one message at one time. Thus, in Nemesis design, even small messages transfer needs atomic enqueue operations.

### 2.2 PGAS Programming Models

Partitioned Global Address Space (PGAS) is an emerging parallel programming model. It’s gathering more and more attentions for the high-productivity feature and better applicability with conjunction of advanced multi-core and network architecture. PGAS provides a shared memory abstraction on distributed memory machines. They also provide control of data layout and work distribution which allows applications developers to take advantage of locality. The PGAS programming model can be implemented either by languages or by libraries. The prominent PGAS languages include: Unified Parallel C (UPC) [102], Titanium [109], Co-Array Fortran [86], X10 [39], Chapel [38], and HPF [77]. There are also
language assisted approaches, such as OpenMPD [73] that are bringing OpenMP style parallelism to distributed memory architectures. Libraries, such as Global Arrays (GA) [56], also implement the PGAS model. The main benefits of language based PGAS approaches are enhanced productivity of application programmers, compiler assisted optimizations and more elegant representation of irregular parallelism. The PGAS programming model and languages are witnessing a renewed interest for Exascale computing.

2.2.1 Unified Parallel C

UPC is one of the most popular PGAS languages based on parallel extensions to the C language. The Berkeley UPC implementation [72] 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 [28] 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 rich, expressive and flexible interfaces that provide 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) [62], mpi (MPI), etc.
2.3 Multi-core and Heterogeneous Architectures

Multi-core architectures are becoming more and more popular in HEC (High End Computing) era. Over the past several years, the number of processor cores on each node has increased dramatically due to the advent of multi-core processing. Systems with 64 cores per node are being deployed, while systems with 128 or even larger number of cores per node are being developed. These latest multi-core architectures, such as AMD Bulldozer and Intel Sandy-Bridge, have been widely adopted in current super computing centers, as well as personal computer systems. Sharing of system resource of multi-core system can increase communication efficiency between processes on the same node. However, it also increases contention of system resource. Furthermore, the rising of accelerators and Co-Processors are increasing the complexity of the contention problems of the modern multi-core and many-core architectures.

2.3.1 GPGPU Clusters

Graphics Processing Units (GPUs) are increasingly being used in the high performance computing systems. In many scientific fields, researchers choose GPGPU clusters for their high peak performance and relatively low cost. In June 2013, there are more than 8% systems in TOP500 which use NVIDIA GPUs to gain significant performance. Special programming framework is necessary for operating GPU devices. The programming framework provided by NVIDIA is called the Compute Unified Device Architecture (CUDA) [87]. Through CUDA, the programmers can write the GPU kernel programs running on GPUs.
GPUs are connected via PCI express in cluster environment. With current hardware support, in order to communicate the data between two GPUs on different nodes, the programmer has to first move the data from device memory to the host memory on the sender side, then send out the data through networks, and finally copy the data from host memory to device memory on the receiver side.

2.3.2 MIC Architecture

Recently, Intel has introduced Intel Many Integrated Core Architecture (MIC) [5], which is a multi-processor computer architecture, to the HPC area. By integrating processors, Intel Xeon Phi coprocessors [7] and accelerators into a relatively small package, the MIC architecture aims to push the HPC area into the exaflop era. Current generation Xeon Phi, which conforms to Intel MIC architecture, is equipped with 61 processor cores 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 a smart scheduler. There are eight memory controllers which can deliver a theoretical bandwidth of up to 352 GB/s. However, with a memory capacity of 8 Gigabytes, the per-core memory limitations place constraints on the number of processes that can be spawned on the MIC. Additionally, MIC offers three modes of execution – offload, native and the symmetric mode [5]. In offload, MIC can be also used in accelerator mode where the application runs only on the host and offloads compute-intensive parts of code to the coprocessor; In native mode, MIC can be used in many-core mode where the application runs only on the Co-Processor; A symmetric mode is also offered on MIC where the application can be launched on both the Co-Processor and the Host.
Current middlewares are taking advantage of the existing multi-core architectures. However, with the emerging of new architectures, such as MIC, new designs and studies need to be applied on these middlewares, in order to achieve the best benefits from the new hardware techniques.

2.4 InfiniBand Architecture

InfiniBand network [3] has been widely used in the high performance computing systems. The InfiniBand Architecture (IBA) defines a switched network fabric for interconnecting compute and I/O nodes. In an InfiniBand network, compute and I/O nodes are connected to the fabric using Channel Adapters (CAs). IBA describes the service interface between a host channel adapter and the operating system by a set of semantics called \textit{verbs}. Verbs describe operations that take place between a CA and the host operating system for submitting work requests to the channel adapter and returning completion status. InfiniBand uses a queue based model. A consumer can queue up a set of instructions that the hardware executes. This facility is referred to as a \textit{Work Queue} (WQ). Work queues are always created in pairs, called a \textit{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 between the consumer’s memory and another consumer’s memory, and the receive work queue holds instructions about where to place data that is received from another consumer. The completion of Work Queue Entries (WQEs) is reported through Completion Queues (CQ).

InfiniBand supports two types of communication semantics: \textit{channel} and \textit{memory} semantics. In channel semantics, the sender and the receiver both must post work request entries (WQEs) to their respective QPs. After the sender places the send work request, the
hardware transfers the data in the corresponding memory region to the receiver end. It is to be noted that the receive work request needs to be present before the sender initiates the data transfer. The sender will not complete the work request until a receive request has been posted on the receiver. This allows for no buffering and zero-copy transfers. When using channel semantics, the receive buffer size must be the same or greater than that of the sending side. Receive WQEs are consumed in the same order that they are posted. In the case of reliable transports, if a send operation is sent on a QP where the next receive WQE buffer size is smaller than needed the QPs on both ends of communication are transitioned into the error state.

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. Both RDMA Write (write to remote memory location) and RDMA Read (read from remote memory location) are supported in InfiniBand, although not all transports support it. RDMA operations allow a node to directly access a remote node’s memory contents without using the CPU at the remote side. These operations are transparent at the remote end since they do not involve the remote CPU in the communication.
Chapter 3: PROBLEM STATEMENT

As discussed in Chapter 1 and Chapter 2, the development of modern multi-core and heterogeneous systems has brought more computation power, as well as more contentions on system and network resources. Meanwhile, emerging applications ask for more powerful support from the middleware, in order to satisfy new requirements such as irregular parallelism, hybrid applications, and so on. Thus, it’s necessary for middleware to not only fulfill the upper-level requirement from different programming models, but also utilize the lower-level hardwares in an efficient manner with less resource contentions. The objectives of this dissertation are to explore and solve both of these aspects of HPC middleware for multi-core and heterogeneous architectures. On the one hand, we deploy new designs and techniques to increase the efficiency of communication substrate; On the other hand, we explore and provide advanced supports in middleware for MPI/PGAS and even hybrid programming models. Figure 3.1 shows the overall scope of this dissertation. Using this framework, we aim to address the following challenges:

1. **Can we reduce the contention on shared memory resources of a modern multi-core system between processes, so that the performance and scalability of inter-process communication within a single compute node can be improved?**

Multi-core systems can provide benefits of low-latency communication through sharing of system resources. On the other side of the coin, sharing can increase contention and
competition over shared resources, such as memory, cache, and even bus bandwidth. As the multi-core systems are widely adopted in HPC area, shared memory channel has been emphasized in current MPI runtime libraries. However, the existing designs are mainly based on a relatively small number of cores/node. As the cluster-on-chip era is emerging, existing shared memory channel designs are facing problems from the dark side of “sharing”.

In order to get prepared for upcoming systems with hundreds of cores/node, it’s necessary for MPI runtime library designers to examine the contention and scalability issues within current shared memory channel designs and propose new solutions to make MPI scaling better on these systems.

2. Can we identify the network throughput congestion of multi-core systems and provide core stateless mechanisms in runtime to control and improve such congestion at node level?
The fundamental premise of this problem is that contemporary or future networks for large scale HPC systems are likely to be underprovisioned with respect to the number of cores or concurrent communication requests per node. In an underprovisioned system, when multiple tasks per node communicate concurrently, it is likely that software or hardware networking resources are exhausted and congestion control mechanisms are activated: the performance of a congested system is usually lower than the performance of a fully utilized yet uncongested system.

We first explore the network throughput congestion issue and prove that on existing multicore systems maximal network throughput cannot be achieved when all cores are active. To improve performance and portability, HPC runtime implementations need to employ novel node level congestion avoidance mechanisms.

3. For multi-threaded runtime, such as multi-threaded UPC runtime, how to utilize the multi-endpoint designs to eliminate network access contentions, and furthermore, provide load balancing options at runtime level, for irregular applications?

With so many cores inside each node, intra-node communication has gained more and more portion in applications. As a result, threads-based runtimes are being emphasized again due to its low-latency. This is because processes-based models have to depend on shared memory or kernel-based schemes for intra-node communications. Furthermore, threads-based runtime can provide low-level load balancing schemes by utilizing the feature of threads: shared address space. Based on the discussion of potential benefits that could be brought by thread-based runtime for multi-core systems, one of the UPC runtime design questions that naturally arises is: On multi-core nodes, should the UPC runtime itself be multi-threaded? This design choice has an impact on several aspects: performance, portability, interoperability and support for irregular parallelism. Past research [54] has
reported that when UPC runtime is implemented using threads, the network performance suffers greatly.

We analyze various design alternatives in-depth for a high performance and scalable UPC runtime in these aspects. Then, in order to achieve the benefits from multi-threaded runtime while getting rid of the network access contention, multi-endpoints design is introduced and implemented inside the multi-threaded UPC runtime. It is also necessary to explore how to utilize the multi-endpoints design for load balancing, in order to better support irregular applications.

4. As heterogeneous systems, such as accelerators and Co-Processor architectures, are getting more and more popular, new resource contention problems are showing up between regular node and GPGPU or MIC. How the middleware adopts new techniques to support different programming models and utilize the hardware resources in a more sufficient way?

Accelerators and Co-Processors are getting huge attention in modern HPC area. The GPGPU and MIC architectures have introduced much more complicated memory levels inside one single compute node. For example, data movement between two GPU devices over network requires several memory copy functions between device and host memory and also involvement of CPUs on both side. Not only is the programming getting complicated, but also new contentions on system resources are getting exposed. In order to fully take advantage of the new architectures, the middleware needs to be enhanced at two levels: 1) provide new support to programming models, such as load balancing, out-of-core data management, global shared address including device memory, to reduce the programming
complexity; 2) incorporate new hardware techniques with existing multi-core and RDMA-based interconnect designs, in order to provide more efficient communication performance and reduced contentions on shared system resources.

The multi-endpoints designs also need to be revisited under the heterogeneous many-core system and multi-GPU environment, in order to better support the unbalanced compute power between accelerators, processors and Co-Processors, as well as the heavier contention on PCI-Express.

5. Can we further optimize the multi-endpoint designs of the runtime in order to achieve better support for hybrid programming models?

MPI has been the de-facto parallel programming model for the past few decades. It is well adopted in various scientific domains and is very successful in implementing regular, iterative parallel algorithms with well defined communication pattern. However, problems with irregular computation and communication patterns are often very challenging for individual application developers to solve in MPI. Instead, PGAS programming models show promise for expressing this kind of algorithms. Meanwhile, the real-world applications depend on many parallel numerical libraries which have been written in MPI. At the same time, the emerging heterogeneous architectures introduce additional programming models: CUDA, OpenCL or OpenACC.

Both of these trends in applications and architectures have increased the interest of hybrid programming models. As a result, it’s the responsibility of the middleware to provide supports to multiple processes/threads from different programming models in an integrated manner, with the least contention and conflicts over system resources. We explore novel designs based on the proposed multi-endpoint runtime with various hybrid programming
models, including MPI+OpenMP, MPI+UPC, MPI+UPC+CUDA and so on, in order to achieve a middleware with better performance and scalability.
Chapter 4: IMPROVE MPI INTER-PROCESS COMMUNICATION OVER SHARED MEMORY

In Chapter 3, we listed several challenges in MPI/PGAS runtime design based on the development of modern multi-core and heterogeneous systems. As mentioned in the first problem, sharing of system resource can increase communication efficiency between processes on the same node, as well as increases contention problems. Currently, most MPI libraries are developed for systems with relatively small number of cores per node. On the emerging multi-core systems with hundreds of cores per node, existing shared memory mechanisms for MPI runtimes will suffer from scalability issue, which may limit the benefits gained from multi-core system.

In this chapter, we analyze several bottlenecks in current process-to-process communication over shared memory on multi-core architectures due to the resource contention. We first analyze these problems and then propose a set of new schemes for small message and large message transfer over shared memory. The “Shared Tail Cyclic Buffer” scheme is proposed to reduce the number of read and write operations over shared control structures. The “State-Driven Polling” scheme is proposed to optimize the message polling through dynamically adjusted polling frequency on different communication pairs. Through dynamic distribution of runtime pinned-down memory, the “On-Demand Global Shared Memory Pool” is proposed to bring benefits of pair-wise buffer to large message
transfer and optimize shared send buffer utilization without increasing the total shared memory usage.

The rest of the chapter is organized as follow. In Section 4.1, we first briefly introduce shared memory channel design in MVAPICH2 and MPICH2-Nemesis. After that we propose three scalability issues considering the system with hundreds of cores per node. Then, we introduce three new schemes to address these issues respectively in Section 4.2. Memory usage of the new designs is discussed in Section 4.3. We evaluate the performance of the new schemes in Section 4.4 with micro-benchmarks and applications. We provide discussion and reference to related works in Section 4.5. Finally, we provide a summary of this chapter in Section 4.6.

4.1 Scalability Issues in Existing Designs

Current MPI shared memory channels are designed for relatively small number of cores. Thus, they are suffering from several problems when the multi-core systems scale up. In this section, we examine these scalability issues within the shared memory channel designs of MVAPICH2 and MPICH2-Nemesis as introduced in last section.

4.1.1 Frequent Shared Structure Access

Small message transfer over shared memory involves three basic stages: copy-in to shared buffer, notify the destination (or the destination discovers the incoming data), copy-out from shared buffer. The cost of the first and the third operations is constant and inevitable. The key to a low-latency small message transfer mechanism thus lies on the second operation. We list the sample code of MVAPICH2 and MPICH2-Nemesis small message transfer over shared control structures and examine the frequency of shared control structure access in Figure 4.1(a) and Figure 4.1(b):
Figure 4.1: Pseudocode for Shared Memory Communication in Existing Designs
<table>
<thead>
<tr>
<th></th>
<th>MVAPICH2</th>
<th>MPICH2-Nemesis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sender</td>
<td>Receiver</td>
</tr>
<tr>
<td>WRITE</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>READ</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4.1: Number of Bus Operations Over Shared Control Structure Per Communication

After a detail examination over the whole strategy, we find that for each communication, the senders and the receivers need one read and one write operations over the shared memory, respectively. We also examine the mechanism utilized by Nemesis: for every communication, senders and receivers need two write and one read operations on the shared memory. Together with MVAPICH2, we provide a summary table of the number of bus operations over shared control structure per shared memory access in Table 4.1.

Figure 4.2: Contentions from accessing shared control structure
In a multi-core system with large number of cores/node, the number of shared structure access should be reduced to as few as possible. This is because shared control structure can be accessed by different processor cores simultaneously. As shown in Figure 4.2, the shared structure of sender 1 is in the nearest memory bank to core 1, it can be accessed by receivers on other cores at the same time, which results in more contentions in the memory bus than that of the private structure access. Thus, the bus operation over shared structure results in more overhead that can increase along with number of cores per node. From the point of view of cache, when the sender and receiver reside in different sockets without shared cache, shared structure access will result in cache invalid and RFO (snoop) operations. In current designs, the shared control structures for different pairs can be located in the same cache line. When a shared structure is accessed by multiple processes simultaneously, more number of cache invalidations will happen.

4.1.2 High Latency in Fair Polling

In MPICH2-Nemesis, shared receive queue with atomic access is used. As a result, receivers only have one place to poll with atomic access overhead. In MVAPICH2, pair-wise buffers are used to alleviate the latency overhead from atomic operations, which requires multiple structures to be polled. Though fair polling scheme is straight-forward, it has a poor scalability. Since the number of flags need to be polled increases linearly with the number of local processes, polling for $n - 2$ pairs can be wasted in the worst case, assuming $n$ cores per node. To utilize the benefits of pair-wise queue, a better polling strategy must be explored.
4.1.3 Memory Usage for Large Memory Transfer

Both current designs in MVAPICH2 and MPICH2-Nemesis utilize ‘shared send buffer pool-based’ scheme for large message transfer. This scheme suffers from three problems in the context of systems with large number of cores/node.

4.1.3.1 Contentions accessing shared send buffer pool and shared receive queue

In MPICH2 Nemesis, this contention comes from the atomic operations that have to be applied on enqueueing stage. In MVAPICH2, the atomic operations are alleviated by notification message exchange and flags indicating the status of elements. However, the small message exchange depends on the pair-wise method. There are still existing atomic operations on the shared data structure at the sender side.

4.1.3.2 Possible Waste of Shared Memory Buffer Pool

Although shared send buffer provides dynamic shared memory utilization inside one single process, the sender buffer could not be shared across all processes on the same node. Thus, memory could be wasted if some processes never have intra-node large message to send. Shared send buffer can be extended to all processes on the same node through sharing of send buffer among all processes, as proposed in [48] for extended discussion about MPICH2-Nemesis design. However, simple extensions like these will make the contention problem even worse: not only enqueueing operations need to be protected, even dequeuing cells from the shared send buffer also need to be atomic.

4.2 Designs

In this Section we propose three new schemes for the challenges we have addressed in Section 4.1.
4.2.1 Shared Tail Cyclic Buffer

In Section 4.1.1, we have claimed that the number of shared data structure access should be reduced for a low-latency small message transfer in a large scale multi-core system. In order to achieve this goal, we propose a new scheme – “Shared Tail Cyclic Buffer” as shown in Figure 4.3. This scheme is based on pair-wise buffer with only one extra shared structure used for the control purpose: “Shared Tail” pointer. Meanwhile, every sender and receiver holds a local head pointer and a local tail pointer respectively. The head indicates where the data should be written in or read out, while the tail indicates where the available buffer ends.

![Diagram of Shared Tail Cyclic Buffer Structure](image)

(a) After two writes from sender and one read from receiver

(b) Update Sender Local Tail

Figure 4.3: Shared Tail Cyclic Buffer Structure

Figure 4.3(a) describes a common send/receive process when the sender has written two 128 bytes message into the shared buffer and the receiver has read the first one. Two pieces of control information are also written into shared buffer in front of the real payload: a “flag” which indicates whether new messages have been written; a “size” parameter which indicates the size of the following message. The receiver detects incoming message by polling the “flag” and copies out the data according to the “size” parameter. The receiver
needs to update the “shared tail pointer” according to its local tail after each copy is finished. On the other side, the sender doesn’t need to update its local tail until there isn’t enough buffer between its local head and local tail for a new message.

Figure 4.3(b) illustrates the situation when the sender has to update its local tail. The local head of the sender keeps increasing as long as the sender continues writing data into the buffer. Assume the size of the next message is 2K. Current local head (7K) and local tail (8K) indicate that not enough room is left in the shared buffer. Once this shortage is detected, the sender will update its local tail according to the “shared tail”, to 7K - 1. Then, the available buffer is enough for copying the message in and the communication can continue. When the sender finds out that the updated tail still doesn’t give enough room for the new message, it will truncate the message and send out the message piece by piece. When the head meets updated tail, it means that the shared buffer is totally full. In this case, the sender can either block wait or return and enqueue the request according to higher layer function.

With the new scheme, we are able to reduce the number of bus operations over shared control structure “Shared Tail”. For the receiver, it is one write over Shared Tail each time. For the sender, it is decreased to $1/n$, where $\sum_{i=1}^{n} s_i \geq N$ and $\sum_{i=1}^{n-1} s_i < N$ ( $s_i$ is the size of the message $i$ and $N$ is the size of the shared buffer). Compared with existing MPI runtimes as shown in Table 4.1, the new Shared Tail design is able to significantly reduce extra accesses over shared control structures.
4.2.2 State-Driven Polling Scheme

In order to fully utilize the benefits of “Shared Tail Cyclic Buffer” scheme without suffering from the scalability problem of fair polling (as discussed in Section 4.1.1), we introduce a new “State-Driven Polling” scheme.

In this polling scheme, every destination peer must be included in one and only one of the two sets: Common Set ($Set_{comm}$) and Priority Set ($Set_{prio}$). At the beginning, all destinations reside in $Set_{comm}$ and $Set_{prio}$ is empty. Polling pattern changes between following three states according to current application action:

- **State 0**: Fair Polling. In this state, all peers are included in $Set_{comm}$. The receiver polls every member in $Set_{comm}$ with the same frequency.

- **State 1**: Priority Polling. In this state, some of the peers are moved from $Set_{comm}$ to $Set_{prio}$. The receiver first polls members in $Set_{prio}$ and returns immediately once it gets a new message from any member in $Set_{prio}$. Only when no message is coming from $Set_{prio}$, the receiver will poll $Set_{comm}$ with the same pattern in State 0.

- **State 2**: Priority Only Polling: In this state, receiver only polls members in $Set_{prio}$.

Transitions between the states are decided by several frequencies and thresholds. The frequency of receiving messages based on total polling numbers from a peer $i$ is recorded as $F_i$. When $F_i$ is larger than a preset threshold $Th_{comm\rightarrow prio}$, $i$ will be moved from $Set_{comm}$ to $Set_{prio}$; when $F_i$ is smaller than threshold $Th_{prio\rightarrow comm}$, the peer $i$ will be pushed back from $Set_{prio}$ to $Set_{comm}$. Another frequency $F_{common}$ is recorded for the incoming messages from all peers in $Set_{comm}$, which is used to spring the transition between State 1 and State 2 by comparing pre-set threshold $Th_{s1\rightarrow s2}$. At last, a counter $C_{miss}$ for continuous failed
polling is kept for State 2, in order to make the receiver fall back to State 1 when it is larger than a pre-set threshold \( T_{h_{s2-to-s1}} \).

All the transitions between states are shown in Figure 4.4.

All the thresholds require dedicated tuning. A small threshold can result in too frequent transitions between states, while a large threshold may result in waste of unwanted polling. Currently, we manually sweep the threshold to get the best performance. Automatic setting of these thresholds according to the number of local cores and application pattern is left for future work.

![Figure 4.4: Receiver Polling State Machine](image)

### 4.2.3 On-demand Global Shared Memory Pool - memory efficient large message transfer

In Section 4.1.3, we analyzed the existing problems in the large message transfer over shared memory. To alleviate effects of these two problems at the same time within a large scale SMP environment, we’d like to bring the benefits of pair-wise method to large message transfer while keeping the total memory usage not going to \( O(n^2) \) with the number of local cores, \( n \). We propose a new “On-demand Global Shared Memory Pool” scheme to achieve this goal. This scheme organizes each shared memory pool created by every local
process as chunks. Then, these memory chunks are managed dynamically. They will be distributed across different processes according to the demands of these processes.

As shown in Figure 4.5, every MPI process allocates a shared memory pool, which is accessible to other local MPI processes. Each shared memory pool is divided into several chunks. At the beginning, these chunks have no owner assigned. Every MPI process keeps separate empty queues for all possible local peers, as shown as “0 to 1 head”, “0 to 2 head”, “1 to 2 head” and so on in the figure. When the first time a sender launches a large message transfer to a local peer, it needs to check the local shared memory pool to reserve one free chunk and link this chunk to the queue specialized for its local peer of this message transfer. This chunk will be marked as reserved (shown as shadowed in the figure). Other senders are unable to reserve the same chunk. When the sender uses up its reserved chunks, it can access the shared memory pool again to reserve more chunks as needed. To better utilize the locality of shared memory for less overhead, the senders will first reserve chunks from the shared memory pool created by itself. Only if there is no free chunks left in its own pool,
it will try to reserve chunks in shared memory pool initialized by other processes (referred as remote pool in the following discussions). Senders follow two principles when they try to reserve chunks from remote pools: First, it can only get chunks from a remote pool with low utilization, in case the original starter process of this remote pool is demanding chunks heavily too; Second, the order of checking remote pools is decided by CPU core locality, i.e., senders are tending to reserve chunks in closer memory bank. We also notice that for some applications, large message transfer can happen only in certain stage and with different senders. To avoid a sender reserving too many chunks and wasting them afterwards, a counter is kept to record the utilization of every reserved chunk. When a chunk is under-utilized, it will be returned to the shared memory pool.

Atomic operations are required when a sender accesses the shared memory pool to reserve or release a chunk. However, we’d like to mention that the reserve operation only happens when the current reserved chunks are used out, and the release operation only happens when the utilization of one chunk is under a pre-set threshold.

This proposed design “On-demand Global Shared Memory Pool” can not reduce shared memory footprint. Shared memory pool still needs to be initialized and pinned-down at the beginning. The user can define the size of the pool or it can be decided according to different platforms. However, through the on-demand and dynamic management of shared memory chunks, shared memory can be distributed to processes who have more demands of shared memory, instead of being wasted by processes with less intra-node large message transfer. As a result, shared memory pool is utilized more efficiently. On the other hand, the pair-wise queue header avoids the extra overhead from atomic operation or re-collecting operations, such as in the previous designs we analyzed in Section 4.1.3.
4.3 Discussion about Memory Usage

“Shared Tail Cyclic Buffer” scheme is based on pair-wise buffer design. One issue associated with the pair-wise buffer design is the memory usage scalability: the total size of pair-wise buffers increases as $O(n^2)$ where $n$ is the number of local cores. In our design, “Shared Tail Cyclic Buffer” scheme is only used for small messages. The small message size depends on the core numbers and the memory size on different multi-core platforms. In the following evaluation section, we assign 8K bytes for each pair-wise shared buffer on Trestles TeraGrid cluster at SDSC, where each node has 32 cores with a total of 64GB memory. Only 256KB from 2GB memory per core is used in our scheme for the small message on Trestles TeraGrid cluster. In the future for a node with hundreds of cores, where $O(n^2)$ gives more pressure on memory usage, we can provide the “Shared Tail Cyclic Buffer” scheme for a set of destinations with more communication frequency, while having shared buffer pool for other destinations.

The “On-Demand Global Shared Memory Pool” scheme implements a dynamic management method for the shared memory chunks, through which each sender can get memory chunks from memory pools created by other processes. This scheme can get better memory utilization than the existing shared sender buffer pool method, where each sender can only use the memory created by itself. As a result, better scalability should be obtained on the future nodes with hundreds of cores.

4.4 Performance Evaluation

In this section, we conduct various experiments to evaluate the performance of our design. Evaluations through micro-benchmark are carried out between popular MPI runtimes: Open MPI 1.4.5rc1 [11], MPICH2 Nemesis 1.5a2 (with fastbox enabled), MVAPICH2 1.7
and the new designs. These schemes are shown as OMPI, NEMESIS, MV2 and NEW in the legends. The experiments are carried out on two platforms. The first cluster is Trestles, a dedicated TeraGrid cluster designed by Appro and SDSC. Each compute node contains four sockets, each with a 8-core 2.4 GHz AMD Magny-Cours processor, for a total of 32 cores per node. Each node has 64 GB of DDR3 RAM. The second platform is a quad socket AMD Bulldozer node with 1.4 GHz AMD Opteron processors, which contains totally 64 cores per node and 128 GB of DDR3 RAM. We refer the first cluster as 32-core system and the second platform as 64-core system in all the following discussion. GCC compiler 4.6.1 is used for both clusters and all the MPI runtimes. All micro-benchmarks results are averaged over 1,000 iterations.

Depending on the related position of senders and receivers, communication can happen inside one socket (Intra-Socket) or across two sockets (Inter-Socket). The following evaluations are carried out for both situations.

4.4.1 Micro-benchmark Evaluation

Now we examine the latency performance of the new schemes comparing with existing schemes by OSU latency benchmark. In the micro-benchmark, one sender and one receiver do ping-pong communications through MPI_Send and MPI_Recv functions.

For the 32-core system, in Figures 4.6(a) and 4.6(b), the new “Shared Tail Cyclic Buffer” achieves 0.60 us and 0.71 us for 4 Byte message on Intra-Socket and Inter-Socket, respectively. This is 31%, 20% and 46% better than MVAPICH2, Nemesis and Open MPI for Intra-Socket communication and 33%, 22% and 46% better for Inter-Socket. For the 64-core system, new design can reduce the latency from 0.5 us to 0.38 us for Inter-Socket as shown in Figure 4.6(c).
Figure 4.6: Micro-benchmark Evaluations: One Pair Ping-Pong Latency

Figure 4.7: Micro-benchmark Evaluations: Multi-Pair Message Rate

Figure 4.8: Micro-benchmark Evaluations: One Pair Bandwidth
We evaluate the impact of state driven polling scheme by message rate micro-benchmark. All cores are occupied on the node. One way MPI_Isend and MPI_Irecv are launched with a window size equal to 64. 32 MPI processes are grouped into 16 pairs on the 32-core system; Similarly, 64 MPI processes are grouped into 32 pairs on the 64-core system. We compare the State-Driven Polling scheme with Shared Tail Cyclic Buffer, current MV2, MPICH2-Nemesis, and Shared Tail Cyclic Buffer with fair polling.

“Shared Tail Cyclic Buffer” can improve the latency by more than 30% for one-pair ping-pong tests as shown in latency evaluations. However, as shown in Figure 4.7, the improvement of “NEW W/ Fair Polling” for non-blocking message injection tests is very limited due to overhead from wasted pollings. Fair polling can only achieve less than 50% message rate when comparing with Nemesis, which uses a single receive queue. However, through “State-Driven Polling” scheme, the message rate can be improved dramatically, which is 5X improvement on the 32-core system and 8X on the 64-core system.

Though the focus of the new proposed large message transfer scheme is the memory efficiency, the pair-wise queues reduce the overhead in accessing shared send queues. Also, for very large message that can not fit in shared memory buffer at once, pair-wise queues are able to avoid synchronizations between each segment. The impact of the proposed dynamic memory pool scheme on bandwidth is shown in Figure 4.8. The performance of only “Shared Tail Cyclic Buffer” scheme is also included in the figure as “NEW STC ONLY”. 

Clearly, the effect of “Shared Tail Cyclic Buffer” stops after the small message threshold 4K. We can observe that on the 32-core system, the new design increases single pair point-to-point bandwidth from 1,000 MB/s to 1,700 MB/s comparing to current MV2 and Nemesis. Compared to OMPI, there is around 100 MB/s improvement. On the 64-core
system. “On-Demand Global Shared Memory Poll” improves the bandwidth by average 600 MB/s from current existing designs.

4.4.2 Application Evaluation

We also examine the impact of the new design alternatives for shared memory communication on several application benchmarks, including Graph500 and NAS benchmarks, on the 64-core AMD Bulldozer system as introduced at the beginning of this section. For application evaluations, results are averaged over 10 iterations and standard deviations are within 0.1% when compared to the average value. In Figure 4.9, we set the performance of MV2 as the baseline and normalize the performance of OMPI, NEMESIS and the new design accordingly.

Figure 4.9: Application Evaluation on 64-core AMD Bulldozer System

Figure 4.10: Communication Topology: NAS-CG application with 512 procs and class D
4.4.2.1 Graph500

Graph500 Benchmark Specification is 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. Several versions of referenced benchmark implementation are available on Graph 500 List [16]. We used the MPI version 1.2 for the following evaluation as shown in Figure 4.9. We have the input graph with $2^{22}$ vertices and $2^{24}$ edges. The new design can improve the performance by 18%.

4.4.2.2 NAS Benchmark

NAS benchmark is a popular application benchmark set for High Performance Computing evaluation [84]. We run BT, CG, FT, LU, MG and SP with CLASS D and NPROCS equal to the number of cores in one node as 64. For benchmarks CG and FT, which have a relatively more intensive communication workload according to profiling, the new designs can achieve benefits of 11% and 9%, respectively. For other benchmarks, the improvements are ranged from 2% to 5%.

To further evaluate the impact of the new designs from a cluster point of view, we carried out the most communication intensive benchmark CG on the 32-core system with 128, 256 and 512 processes, that is, 4, 8 and 16 nodes, respectively. As shown in Table 4.2, when the number of processes increases, the impact of the new designs has greatly increased from 2.2% to 16.7%. To explain the impact of shared memory design on an application running with multiple nodes, we profiled the communication pattern of CG application with IPM [4] and the communication topology is shown in Figure 4.10. From the profiling results, we find out that the destinations of most communications are neighboring
<table>
<thead>
<tr>
<th>Num. of Procs</th>
<th>MV2</th>
<th>NEW</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>199.06</td>
<td>194.55</td>
<td>2.2%</td>
</tr>
<tr>
<td>256</td>
<td>150.61</td>
<td>139.73</td>
<td>7.2%</td>
</tr>
<tr>
<td>512</td>
<td>104.02</td>
<td>86.64</td>
<td>16.7%</td>
</tr>
</tbody>
</table>

Table 4.2: CG (CLASS D) Performance on Trestles (Time: sec)

processes. As a result, when we place rank processes linearly on the nodes, most communications reside inside the same node. The communication pattern is same for 128/256 processes and we skip these figures for brevity. Furthermore, when the number of processes involved in the application increases from 128 to 512, the portion of MPI communications keeps increasing. Since the inter-node communication performance also impacts the final results, we only include the current MV2 and the new designs, which use the same inter-node mechanism, in this comparison. The trend clearly shows that the performance of intra-node communication has a great impact on HPC applications, with heavy message exchange between closer ranks. With the scale of multi-core system increasing, we believe this impact will be broadened to more applications.

4.5 Related Works

In this chapter, we have discussed the brief idea of the shared memory channel design in current MVAPICH2 and MPICH2-Nemesis based on the research proposed in [33, 34, 37].

Kernel-based direct memory access, such as LiMIC [66] and KNEM [79], is another way to by-pass shared memory overhead, which uses OS kernel to map memory address from one process to another process and do one memory copy from send buffer to receive buffer directly. This method can improve the performance of medium and large messages.
as shown in [83]. However, kernel-based memory transfer faces multiple problems such as security issue, and high overhead for small message transfer.

On multi-core system, many studies have explored the contention problem. Hood et al. [61] used a differential performance analysis methodology to quantify the costs of contention for resources in the memory hierarchy of several multi-core processors used in high-end computers. Jin et. al evaluated HPCC and NAS benchmarks through similar analysis methodology on different platforms in [60]. Majo et al. [81] explored the counteraction between optimization of data locality and cache contention avoidance. These works revealed the relationship between applications performance and resource contention on multi-core systems, but didn’t provide a general solution. Our method provides a solution for cache contention avoidance in MPI library. Zhuravlev et al. provided a comprehensive analysis of contention-mitigating techniques and proposed contention-aware scheduling on UMA architecture in [112]. Blagodurov et al. proposed a contention-aware scheduling algorithm for NUMA systems in [26], along with investigating memory migration strategies. However, the scheduling methods presented in these papers are not a standard part in current OS kernel. Compared with the scheduling optimization at OS level, our method doesn’t require any kernel module or kernel modification.

4.6 Summary

In this chapter, we examined the current shared memory channel design in two popular MPI stacks: MVAPICH2 and MPICH2-Nemesis. Based on the three scalability issues within current designs, we proposed three new schemes for MPI shared memory communication: the “Shared Tail Cyclic Buffer” is aimed at reducing extra access to shared control structures; the “State-Driven Polling” scheme is used to reduce polling overhead with more
and more local destinations for pair-wise buffers; the “On-Demand Global Shared Buffer Pool” is proposed to reduce contentions in accessing shared send buffer and shared receive queues and provide an efficient memory usage for large message transfer. These new schemes can bring up to 26% improvement for point-to-point latency benchmark and 70% improvement for point-to-point bandwidth benchmark. For NAS and Graph500 application benchmarks, the new schemes can achieve up to 15% improvement on the 64-core/node Bulldozer system and 16% improvement on the 32-core/node Mangy-Cours system.
Chapter 5: CONGESTION AVOIDANCE ON MULTI-CORE HIGH PERFORMANCE COMPUTING SYSTEMS

Efficient communication is a requirement for application scalability on High Performance Computing systems. In last chapter, we focused on contention problem inside multi-core systems. However, as the number of cores/node keeps increasing, contention does not only happen with communications inside the node, but also with communications going outside each node. We continue with the second problem proposed in Chapter 3, which is related to the congestion issue for communications from multi-core systems into network.

In this chapter we argue for incorporating proactive congestion avoidance mechanisms into the design of communication layers on many-core systems. This is in contrast with the status quo which employs a reactive approach, e.g. congestion control mechanisms are activated only when resources have been exhausted. We present a core stateless optimization approach based on open loop end-point throttling, implemented for two UPC runtimes (Cray and Berkeley UPC) and validated on InfiniBand and the Cray Gemini networks. Our runtime provides both performance and performance portability. Overall, our research indicates that modern systems accommodate only a surprisingly small number of messages in flight per node. As Exascale projections indicate that future systems are likely to contain
hundreds to thousands of cores per node, we believe that their networks will be underprovisioned. In this situation, proactive congestion avoidance might become mandatory for performance improvement and portability.

The rest of this chapter is organized as follows. In Section 5.1 we describe microbenchmarks to understand the variation of network performance with the number of messages in flight, to recognize congestion and to derive heuristics used for message throttling. Then, in Section 5.2, we present several designs for congestion avoidance mechanisms implemented in two Unified Parallel C [102] runtimes on different networks: the Berkeley UPC [72] implementation on InfiniBand and the Cray UPC implementation on Cray XE6 systems with the Gemini interconnect. In Section 5.2.1, we discuss the design of a message admission control policy using rate or count based metrics. In Section 5.2.2, we present inline and proxy based implementations of the admission control policy. We demonstrate how the proposed congestion avoiding runtime is able to provide both performance and performance portability for applications in Section 5.3. In Section 5.5, we discuss related work. Finally, we conclude this chapter in Section 5.6.

5.1 Network Performance Characterization

In order to understand the variation of network performance with the number of messages in flight, to recognize congestion and to derive heuristics used for message throttling, in this section we carry out a suite of UPC microbenchmarks that vary: i) the number of active cores per node; ii) the number of messages per core; iii) the number of outstanding messages per core; and iv) the message destination. We consider bi-directional traffic, i.e. all cores in all nodes perform communication operations and we report the aggregate bandwidth. We have performed experiments where each core randomly chooses a destination
for each message, as well as experiments where each core has only one communication partner. Both settings provide similar performance trends and in the rest of this chapter we present results only for the latter.

5.1.1 Experimental Setup

We use two large scale HPC systems for our evaluation. Trestles is a 324 compute nodes cluster at the San Diego Supercomputing Center. Each compute node is quad-socket, each with a 8-core 2.4 GHz AMD Magny-Cours processor, for a total of 32 cores per node and 10,368 total cores for the system. The compute nodes are connected via QDR InfiniBand interconnect, fat tree topology, with each link capable of 8 GB/s (bidirectional). Trestles has a theoretical peak performance of 100 TFlop/s.

NERSC’s Cray XE6 system, Hopper, has a peak performance of 1.28 Petaflops/sec and 153,216 cores organized into 6,384 compute nodes made up of two twelve-core AMD ‘MagnyCours’. Hopper uses the Cray ‘Gemini’ interconnect for inter-node communication. The network is connected in a mesh topology with adaptive routing. Each network interface handles data for the attached node and relays data for other nodes. The “edges” of the mesh network are connected to each other to form a “3D torus.” The Gemini message latency is \( \approx 1 \mu s \) and two 24 core compute nodes are attached to the same NIC, thus 48 cores share one Gemini card.

All the software described in this chapter is implemented as a thin layer interposed between applications and runtimes for the Unified Parallel C (UPC) language. On the InfiniBand network we use the Berkeley UPC runtime [72], version 2.12.2. BUPC is free software and it uses the GASNet [28] layer for communication which provides highly optimized one-sided communication primitives. In particular, on InfiniBand GASNet uses
the OpenIB Verbs API. On the Cray system, we use the Cray UPC compiler, version 5.01 within Cray Compiling Environment (CCE) 7.4.2. The Cray UPC runtime is built using the DMAPP\textsuperscript{1} layer. We also experiment with MPICH, Cray MPI and OpenMPI.

\subsection*{5.1.2 Analysis}

Figure 5.1(a) shows the aggregate node bandwidth on InfiniBand when increasing the number of cores. Each core uses blocking communication and we present three runtime configurations (‘Proc’, ‘Hyb’ and ‘Pth’) that are characterized by increasing message injection overheads, as first indicated by Blagojevic et al. [27]. The series labeled ‘Proc’ shows results when running one process per core, the series labeled ‘Hyb’ shows one process per socket with \texttt{pthreads} within the socket and the series ‘Pth’ shows one \texttt{pthread} per node.

In general, best communication performance [27] is obtained when threads within a process are mapped on the same socket, rather than spread across multiple sockets. For lack of space, we only summarize the performance trends without detailed explanations.

For all message sizes, the throughput with ‘Pth’ keeps degrading when adding more sockets. With ‘Hyb’, the throughput slightly increases up to two sockets active, after which it reaches a steady state. With ‘Proc’, which has the fastest injection rate, the throughput increases up to two sockets, after which it drops dramatically. In the best configuration, ‘Proc’ has 3X better throughput than ‘Hyb’ and 15X better throughput than ‘Pth’. The performance difference between best and steady state ‘Proc’ throughput is roughly 2X. Similar behavior is observed across all message sizes.

These trends are a direct result of congestion in the networking layers. When running \texttt{pthreads}, runtimes such as BUPC or MPI use locks to serialize access to the networking hardware: the larger the number of threads, the higher the contention and the higher the

\textsuperscript{1}Distributed Memory Application API for Gemini
message injection overhead. The UPC ‘Proc’ configuration running with one process per core\(^2\) on InfiniBand, does not use any locks to mitigate the network accesses and the drop in throughput is caused by either low level software (NIC driver) or hardware. Since one process per core provides the best default performance for UPC and it is the default for MPI, the results presented in the rest of this chapter are for this particular configuration.

Figure 5.1(a) indicates that there is a temporal aspect to congestion and throughput drops when too many endpoints inject traffic concurrently. We refer to this as *Concurrency Congestion* (CC) and informally define its threshold measure as the number of concurrent transfers from distinct endpoints (with only one transfer per endpoint) that maximize node bandwidth. For example, any ‘Proc’ or ‘Pth’ run with more than 20, respectively four, cores active at the same time is said to exhibit *Concurrency Congestion*. On Cray Gemini, congestion is less pronounced and it occurs when more than 40 of the 48 cores per NIC are active. We believe that ours is the first study to report this phenomenon.

Avoiding CC requires throttling the number of active endpoints and Figure 5.1(b) shows the performance expectations of a simple optimization approach. Assuming \(N\) cores per node, there are \(N\) messages to be sent and we plot the speed-up when using only subsets of \(C\) cores to perform the communication. There are \(\frac{N}{C}\) rounds of communication and we plot \(\frac{T(N) - T(C) + \frac{N}{C}}{T(N)}\). Positive values on the \(z\) axis indicate performance improvements. For example, on InfiniBand using 16 cores to send two stages of 16 messages each is up to 37\% faster than allowing each core to send its own message.

Optimized applications use non-blocking communication primitives to hide latency with communication-communication and communication-computation overlap. Figure 5.2 shows results for a two node experiment where we assume each core has to send a large

\(^2\)One thread per process, rather.
(a) Variation of node throughput with the number of active cores

(b) Throughput Improvement when Restricting Active Cores: InfiniBand

Figure 5.1: Concurrency Congestion: node throughput drops when multiple cores are active at the same time. We assume each core has one message to send and we plot the speedup of using cores/x rounds of communication over all cores active.

(a) Throughput Variation with Msg/Core - InfiniBand

(b) Throughput Variation with Msg/Core - Gemini Band

Figure 5.2: Rate Congestion: node throughput drops when cores have multiple outstanding messages. We assume each core has to send 1,024 messages and we plot the speedup of using 1,024/x rounds of communication with x = (1,2,4..) messages over sending 1,024 messages at the same time.
Figure 5.3: Concurrency and Rate Congestion in MPI. Note that MPI (two-sided) exhibits even larger performance degradation, up to 600%, than UPC (one-sided).

number of messages. The figure plots the relative differences between communication strategies: $N$ outstanding messages compared with $\frac{N}{D}$ rounds of communication, each with $D$ outstanding messages: $\frac{T(D) + \frac{N}{D} T(N)}{T(N)}$. In the first setting, each core initiates $N$ non-blocking communication requests then waits for completion, while in the second, it waits after initiating only $D$ requests.

Intuitively, congestion occurs whenever performing multiple rounds of communication is faster than a schedule that initiates all the messages at the same time. On the InfiniBand system this occurs whenever there is more than one message outstanding per core and best throughput is obtained using blocking communication. Increasing the number of outstanding messages per core decreases throughput, e.g. two outstanding messages per core deliver half the throughput of blocking communication. Asymptotically, the throughput difference is as high as 4X. The Cray Gemini network can accommodate a larger number of messages in flight and there is little difference between one and two outstanding messages per core;
with more than two messages per core throughput decreases by as much as 2X. These results indicate that when all cores within the node are active, best throughput is obtained when using blocking communication.

On both systems, reducing the number of active cores determines an increase in the number of outstanding messages per core that provides best throughput, e.g. on InfiniBand with one core active, best throughput is observed with 40 outstanding messages.

Intuitively, the behavior with non-blocking communication illustrates the spatial component of congestion, i.e. there are limited resources in the system and throughput drops when space is exhausted in these resources. For the purpose of this study, we refer to this as Rate Congestion. As with CC, we define the threshold for Rate Congestion as the number of outstanding messages per node that maximizes throughput. Note than while CC distinguishes between the traffic participants, RC does not impose any restrictions.

The behavior reported for the UPC microbenchmarks is not particular to one-sided communication or caused solely by implementation artifacts of GASNet or DMAPP. Similar behavior is shown in Figure 5.3 for MPI on both systems. Note the very high speedup (250% on InfiniBand and 700% on Gemini) of MPI throughput when restricting the number of cores: Overall, it appears that MPI implementations exhibit worse congestion than the UPC implementations.

Increasing the number of nodes participating in traffic and varying the message destinations does not change the trends observed using a two node experiment. For lack of space we do not include detailed experimental results but note that on both systems, best throughput when using a large number of nodes is obtained for workloads that have a similar or lower number of outstanding messages per node than the best number required for two nodes. Workloads with small messages tend to be impacted less at scale than workloads
containing large messages. In summary, congestion happens first in the Network Interface Card, with secondary effects inside the network when using large messages.

5.2 Core Stateless Congestion Avoidance

As illustrated by our empirical evaluation, network throughput drops when tasks initiate too many communication operations. We interpose a proactive congestion avoidance layer between applications and the networking layer that allows traffic to be injected into the network only up to the threshold of congestion.

Our implementation redefines the UPC level communication calls and it is transparently deployed for the BUPC and the Cray runtimes. While the UPC language specification allows only for blocking communication, e.g. upc_memget, all existing implementations provide non-blocking communication extensions. Avoiding Concurrency Congestion requires instrumenting the blocking calls, while avoiding Rate Congestion requires instrumenting the non-blocking calls.

The first component is an admission control policy that determines whether a communication request can be injected into the network. In order to provide scalability, we explore core stateless policies: communication throttling decisions are made using open-loop control with only task or node knowledge and without any information about the state of the outside network. In contrast, previous work [30, 43, 89] tries to correlate MPI Send and Receive events. We derive the congestion thresholds and heuristics to drive the admission control policy from the results reported by the microbenchmark described in Section 5.1.

For each communication call, e.g. upc_memget, our implementation consults the admission control policy. To address both Rate and Concurrency Congestion we present a

\[^{3}\)No monitoring or feedback loop.
count based policy that uses node level information for message access. Mostly for comparison with delay based techniques (and for the few scenarios where node level network state information is not available to endpoints), we design a rate based policy that allows each endpoint to inject traffic based only on knowledge about its own history. While providing the most scalable runtime design, rate based admission is expected to be able to avoid only Rate Congestion.

We have implemented the admission control policy using multiple designs. In the inline design, each task is directly responsible for managing its own communication operations, e.g. the task that initiates a \texttt{upc\_memget} is also the only one capable of deciding it has completed. Note that this is the functionality implicitly assumed and supported by virtually all contemporary communication layers and runtimes. In the proxy based design, a set of communication “servers” manage client requests. We provide a highly optimized implementation that uses shared memory between tasks (even processes) and allows any task to initiate and retire any communication operation on behalf of any other task. To our knowledge, we are the first to present results using this software architecture within a HPC runtime. The approach is facilitated by GASNet, which provides a communication library for PGAS Languages. We wish to thank the GASNet developers for providing the modifications required to enable any task to control any communication request.

### 5.2.1 Admission Control Policies

**Count Limiting:** In the count based approach, each node has a predetermined set of tokens. Any task has to acquire enough tokens before being allowed to call into the low level communication API. The number of tokens can either be fixed, \textit{i.e.} one token per request, or can be dynamically specified based on the message size; larger messages are associated
with more tokens. Completion of a request involves relinquishing the tokens consumed at posting. Given that we provide a single token pool per node, unfairness is certainly a concern as it reduces the performance of SPMD programs by increasing the synchronization time. To reduce the likelihood of such behavior, we use a ticket-based token allocation that guarantees a first-come first-serve policy. Specifically, threads that are denied network access are given a numbered ticket with the tokens requested. Completion of requests is associated with activation of the next ticket in-line.

We have explored both one token per message and “size proportional” allocations. Achieving optimal performance requires to use for each message a number of tokens proportional to its size. We have implemented a benchmark that performs a guided sweep of different token allocation strategies and synthesizes a total number of tokens per node as well as the number of tokens required by any size. As we explore a multidimensional space \([\text{nodes, cores per node, msgs per core, msg size}]\), this is a very labor intensive process which we plan to automate in future work. The parameters determined by the offline search are then used to specialize the control avoidance code.

**Rate Limiting:** In the rate limiting approach, tasks are not allowed to call into the low level API for certain periods of time in an attempt to throttle the message injection rate. If the time difference between the last injection time-stamp and the current time is less than a specified threshold, the thread waits spending its time trying to retire previously initiated messages. To determine the best self-throttling injection rates, we implemented a benchmark that for each message size sweeps over different values of injection delay using a fine-grained step of 0.5 µs. For each message size we select the delay that maximizes node throughput.
Figure 5.4: Software architecture of the Proxy implementation. An admission control policy layer can be easily added behind the servers.

5.2.2 Admission Control Implementations

**Inline Admission Control:** We implement two variants of inline enforcement of the admission control policy. The first variant, which is referred to in the rest of this chapter as *inline rate throttling* (IRT) uses rate control heuristics based on task local knowledge with synchronous behavior with respect to message injection. IRT is provided mostly for comparison with the related optimization [30, 43, 89] approaches. The second variant, referred to as *inline token throttling* (ITT) uses count limiting heuristics based on node-wide information and it has synchronous behavior with respect to message injection. We did not implement IRT with node knowledge due to the lack of synchronized node clocks.

**Proxy Based Admission Control:** We implement a proxy based admission control that provides non-synchronous behavior with respect to message injection at the application level. The design is presented in Figure 5.4. We group tasks in pools and associate a communication server with each task pool. Besides being a client, each application level task can act as a server. Each task has an associated request queue and whenever it wants
to perform a communication operation, it will place a descriptor in its “Per Task Queue”. Afterwards, the task tries to grab the “Server Token” associated with its pool. If the token is granted, the task starts acting as a server, polls all the queues in the pool and initiates and retires communication operations. When a task masquerades as a server, it serves queues in a round-robin manner, starting with its own. This approximates the default best-effort access to the NIC provided by the underlying software layers. If a token is denied, control is returned to the application and the request is postponed.

Another possibility which still has to be fully tested is to have a server issue all the messages within a queue before proceeding to the next queue. This strategy has the effect of minimizing the number of active routes when a task has only one communication partner within a “scheduling” window.

Such a software architecture is able to avoid both types of congestion. In addition, having the requests from multiple tasks aggregated into a single pool increases the opportunity of more aggressive communication optimizations such as message coalescing or reordering. Concurrency Congestion is clearly avoided by controlling the number of communication servers. The question remains whether the overhead and throttling introduced by the proxy indirection layer is able to prevent Rate Congestion by itself or supplementary avoidance mechanisms are required behind the server layer. Rate Congestion avoidance might require controlling the number of requests in flight per server.

In the experimental evaluation section, this implementation is referred to as Proxy.
Figure 5.5: Performance impact of Rate (IRT), Token (ITT) and Proxy congestion avoidance on InfiniBand. We plot the speedup of applying our congestion avoidance while increasing the number of outstanding messages per core (1, 2, 4, 8, 16).

Figure 5.6: Performance impact of Token (ITT) and Proxy congestion avoidance on Gemini. We allow an increasing number of outstanding messages per core and plot the speedup of applying our congestion avoidance mechanisms.
Figure 5.7: Overhead of Congestion Avoidance with Active Cores. We plot the speedup for an increasing number of active cores, number of messages per core and message sizes. We vary the message size from 8B to 512KB in increasing powers of 2 and there is one data point per 8, 16, ..., 512KB.

5.3 Evaluations

5.3.1 Microbenchmark Evaluation

Figure 5.5 shows the performance when running the microbenchmark described in Section 5.1 on top of our congestion avoiding runtime on InfiniBand. We plot the speedup of congestion avoidance over the default runtime behavior. The parameters controlling the behavior of congestion avoidance are obtained using sweeps as described in Sections 5.2.1 and 5.2.2.

Figure 5.5(a) shows the impact of IRT for microbenchmark settings with an increasing number of operations per task, i.e. for the series “2” each task issues two non-blocking operations at a time. As expected, IRT it is not able to avoid Concurrency Congestion (series “1”). When tasks issue a larger number of transfers IRT provides a maximum of 2X performance improvements. The largest improvements are observed for messages shorter than 2KB.
Figure 5.5(b) shows that ITT avoids both types of congestion and we observe as much as 3X speedups. Note that the 50% speedup is obtained when throttling blocking communication.

Figures 5.5(c) and (d) show the impact of the Proxy implementation for settings where we allow one and 256 outstanding operations per core. We plot the speedup obtained when using two, four, eight and 16 servers per node and observe speedups as high as 1.6X. When the degree of communication concurrency per core is low, e.g. blocking communication, Proxy performs best when the number of active servers (16) is close to the concurrency congestion threshold. When the communication concurrency per core is high (256 outstanding messages), Proxy by itself cannot prevent Rate Congestion and best performance is obtained with two servers. The series “8 serves + ITT” shows that implementing an additional admission control layer behind the servers enables the Proxy design to handle both Rate and Concurrency Congestion.

Figure 5.6 shows the performance on Gemini. ITT is able to improve performance whenever tasks issue more than two outstanding messages and by as much as 5X when issuing 64 outstanding messages. The Proxy implementation improves performance for workloads with at least four outstanding messages per core.

Our approach delays message injection and it might decrease throughput when the communication load is below the congestion thresholds. Figure 5.7(a) and Figure 5.7(b) show the impact of unoptimized ITT on Gemini and unoptimized Proxy on InfiniBand throughput when increasing the number of active sockets per node and the number of messages per core. In this case, we are using a predictor independent of the message size, i.e. one token per message. For short messages, ITT on Gemini decreases throughput by at most 10% independent of core concurrency. The data indicates that, although it introduces
only a low overhead, it is beneficial to disable our congestion avoidance mechanism for certain message sizes when only a subset of cores is active.

### 5.3.2 Building a Congestion Avoidance Policer

The microbenchmarks presented throughout this chapter indicate that congestion avoidance should be driven by the following control parameters: i) concurrency congestion threshold ($CCT[\text{size}]$) measured as the number of cores that when active decrease throughput of blocking communication; ii) node congestion threshold ($NRCT[\text{size}]$) measured as the total number of outstanding messages per node that “maximizes” throughput; iii) core congestion threshold ($CRCT[\text{active\_cores}][\text{size}]$) measured as the number of non-blocking operations per core that “maximize” throughput at a given core concurrency. Intuitively, these parameters capture the minimal amount of communication parallelism required to saturate the network interface card.

```plaintext
Input: size, dest
if active\_cores < CCT[\text{size}] then // no concurrency congestion
    if active\_node\_msgs < NRCT[\text{size}] AND active\_msg < CRCT[active\_cores][\text{size}]
        then
            // no congestion
            inject(size, dest)
        else
            // rate congestion detected
            avoid\_RC(size,dest)
    end
else
    // concurrency and rate congestion detected
    avoid\_CC\_RC(size,dest)
end

Algorithm 5.1: Pseudocode for Congestion Avoidance
```
Algorithm 5.1 shows the pseudo code for the control decisions in our mechanism. We give priority to dealing with Concurrency Congestion and then we try to avoid Rate Congestion. The “procedures” use internally a count based predictor for both ITT and the Proxy implementation. For Proxy we add an admission control layer behind the servers. All parameters, including the tokens per node, are determined by iteratively executing the microbenchmarks using manual guidance. At this point, the predictor we synthesize in practice contains thresholds that are independent of the message size, i.e. one token per message. For Proxy, we also search for the best server configuration. Our results on the InfiniBand cluster indicate that eight servers per node produce good results in practice, while on Gemini, 24 servers per node are required. This amounts to a ratio of four, respectively two tasks per server.

The behavior on InfiniBand using the tuned predictors is shown in Figure 5.7(c). When varying the number of active cores, messages per core and size of the message, our implementation improves performance in most of the cases. In very few cases it introduces a small overhead, at most 4%. We are still investigating the behavior of ITT when using eight cores with blocking communication. Similar trends are observed on Gemini. When comparing with Figures 5.7(a) and 5.7(b) which show unoptimized ITT and Proxy performance, tuning reduces drastically the number of configurations where performance is lost. For the configurations where our implementation actually slows down the microbenchmark execution, the predictor causes an average slowdown of 2% across varying core concurrencies, message sizes and messages per core.

As we have only partially processed a large volume of experimental data, we believe that we can further tune the predictors and improve the performance of our mechanisms.
5.3.3 All-To-All Performance

![Graphs showing All-To-All Performance](image)

Figure 5.8: Impact of IRT, ITT and Proxy congestion avoidance on all-to-all performance. We plot the speedup of MPI and that of single implementation running on top of congestion avoidance (IRT, ITT, Proxy). The performance baseline is an overlapped implementation in UPC.

All-to-all communication is widely used in applications such as CPMD [1], NAMD [90], LU factorization and FFT. MPI [69, 101] and parallel programming languages such as UPC [85] provide optimized implementations of all-to-all collective operations. Most if not all of the existing implementations use multiple algorithms selected by message size. Bruck’s algorithm [31] is used for latency hiding for small messages and it completes in $\log(P)$ steps, where $P$ is the number of participating tasks. For medium message sizes an implementation overlapping [101] all the communication operations is used. In this
implementation, tasks use non-blocking communication and initiate $P - 1$ messages. For large message sizes, a pairwise exchange [101] algorithm is used where pairs of processors “exchange” data using blocking communication.

To demonstrate the benefits of our congestion avoidance runtime, we compare the performance of a single algorithm all-to-all against the performance of library implementations. Our baseline implementation is the overlapping “algorithm” with each processor starting to communicate with $MYTHREAD + 1$. In Figure 5.8, we plot the speedup of multiple implementations over the baseline implementation running on the native UPC runtime layers. For reference, the series labeled MPI presents the performance of the MPI_Alltoall on the respective system. The performance of the UPC library all-to-all is similar to MPI and not shown. On both systems the library calls implement Bruck’s algorithm for small messages. The series labeled “exchange-pw” presents the performance of a handwritten pairwise exchange implementation in UPC. We have also implemented pairwise exchanges in MPI, the results are similar to exchange-pw and omitted for brevity. The series labeled “ITT” and “Proxy” show the performance of the overlapping algorithm with a runtime that implements ITT and Proxy congestion avoidance respectively. These implementations are not tuned and use a simple count based predictor enabled for all core concurrencies and message sizes. The series labeled “Tuned Proxy” shows the behavior of a tuned implementation of Proxy and it illustrates the additional benefits after a significant effort to mine the experimental data.

On the InfiniBand network, “Tuned Proxy” provides best performance and we observe speedups as high as 90% and 170% for 512 byte messages on two nodes and 1,024 cores, respectively. Furthermore, our implementation is faster than any all-to-all deployed on the system for medium to large messages. For example, “Tuned Proxy” is roughly 5X faster
than the MPI library at 1KB messages. We omit any rate throttling results (IRT) since IRT provides only modest performance improvements.

On Gemini, our congestion avoiding runtime provides again the best performance. The MPI library is not as well tuned on Gemini and our implementation is as much as 6X faster than MPI for medium sized messages. ITT provides better performance than Proxy and IRT provides the least improvements. Except for small messages where MPI uses Bruck’s algorithm, ITT is faster than any communication library deployed on the Cray, by as much as 25% for 2KB messages when using 768 cores.

These results indicate that our congestion avoiding runtime is able to improve performance and provide performance portability. We have obtained best performance on two systems using one implementation when compared against multi-algorithm library implementations.

5.3.4 Application Benchmarks

![Figure 5.9: RandomAccess on 1,024 cores InfiniBand. We plot the speedup relative to a baseline implementation using blocking communication.](image)

![Figure 5.10: The NAS Parallel Benchmarks on InfiniBand. We plot the speedup relative to a baseline implementation using blocking communication.](image)
We evaluate the impact of our congestion avoiding runtime on several application benchmarks written by outside researchers. The HPCC RandomAccess benchmark [21] uses fine grained communication, while the NAS Parallel Benchmarks [17] are optimized to use large messages. Fine-grained communication is usually present in larger applications during data structure initializations, dynamic load balancing, or remote event signaling.

The current UPC language specification does not provide non-blocking communication primitives and all publicly available benchmarks use blocking communication. On the other hand, both BUPC and Cray provide nonblocking extensions. We have modified each benchmark implementation to exploit as much communication overlap as possible. All the performance models and heuristics described in this chapter have been implemented in a thin layer between the application and the runtimes for Berkeley UPC and Cray UPC, which is transparent to the application developer.

**RandomAccess:** The RandomAccess benchmark is motivated by a growing gap in performance between processor operations and random memory accesses. This benchmark intends to measure the peak capacity of the memory subsystem while performing random updates to the system memory. The benchmark performs random read/modify/write accesses to a large distributed array, a common operation in parallel hash table construction or distributed in-memory databases. The amount of work is static and evenly distributed among threads at execution time. Figure 5.9 presents the results on InfiniBand when using 1,024 cores. We plot the speedup relative to a baseline implementation that uses only blocking communication primitives. The x-axis plots the number of indirect references per thread. The message size for every single operation is 16 byte. The first three bars (IRT-block, ITT-block, and Proxy-block) plot the speedup observed when running the baseline implementation with congestion avoidance and illustrate the capability of our runtime to
avoid *Concurrency Congestion*. Proxy is able to provide speedup as high as 57%. The series labeled “nb” plots the performance of a hand optimized implementation in which the inner loops are unrolled and communication is pipelined and overlapped with computation and other communication. This is the de facto communication optimization strategy that is able to improve performance by as much as 40%. The series IRT-nb, ITT-nb and Proxy-nb show the additional performance improvements of congestion avoidance for *Rate Congestion* and Proxy-nb is able to provide as much as 60% speedup. As indicated by Figure 5.2, the small messages in RandomAccess do not generate congestion on Gemini and our runtime does not affect its performance.

The behavior of RandomAccess illustrates an interesting performance inversion phenomenon: an implementation with blocking communication and congestion avoidance is able to attain better performance than an implementation hand optimized for communication overlap. Best performance is obtained by the implementation optimized for overlap and using congestion avoidance.

**NAS Benchmarks:** All implementations are based on the official UPC [17, 65] releases of the NAS benchmarks, which we use as a performance baseline. For brevity, we do not provide more details about the NAS Parallel Benchmarks, for a detailed description please see [23, 57]. The benchmarks exhibit different characteristics. FT and IS perform all-to-all communication. SP and BT use scatter-gather communication patterns. SP issues requests (Put) to transfer a variable number of mid-size contiguous regions. The requests in BT (Put and Get) vary from small to medium sizes. In MG, the communication granularity varies dynamically at each call site. CG uses point to point communication with constant message sizes. For all benchmarks, the count and granularity of messages varies with problem class and system size. Vetter and Mueller [103] indicate that large scientific applications
show a significant amount of small to mid-size transfers and all the benchmark instances considered in this chapter exhibit this characteristic.

Figure 5.10 presents the results on InfiniBand. As discussed, the implementations that use `upc_memput` show no performance improvement. Best performance improvements are observed for the communication intensive benchmarks (CG, FT, IS) and we observe as much as 17% speedup for IS.

### 5.4 Discussion

Most of the previous work [3,47,55,67,70,107,108,110] address congestion in the core (switches) of HPC networks. As our experimental evaluation shows, the advent of multi-core processors introduces congestion at the edge of these networks and mechanisms to handle *Concurrency Congestion* are required for best performance on contemporary hardware. Our count based heuristic can handle both *Rate* and *Concurrency Congestion* and it has been easily incorporated into software architectures using either task level (ITT) or node level (Proxy) mechanisms. While ITT is simple to implement and provides good performance, we favor in the long run the Proxy with ITT design which allows for further optimizations such as coalescing and reordering of communication operations. We also believe that the admission control policy heuristics can be further improved.

All the experimental results illustrate the challenges of writing performance portable code in a multi-system, hybrid-programming model environment and we have shown that our congestion avoiding runtime provides both performance and performance portability. The current optimization dogma advocates for exposing a large concurrency and hiding latency (with multi-threading or other optimizations) by overlapping communication with other work. Our experiments indicate that each system supports only a very limited amount
of communication concurrency without significant performance degradation. Our techniques allow developers to expose the maximum “logical” concurrency at the application level and throttle it at runtime for optimal performance. Also, note that without congestion avoidance, our evaluation indicates that overlap is becoming harder to achieve with portability on manycore systems.

Examining the design tradeoffs of congestion avoidance mechanisms, and in general application optimization tradeoffs, we see two main design criteria: 1) optimizing for overlap; and 2) optimizing for throughput. As overlap requires fast message injection and throughput requires throttling and delays, these two have contradictory requirements. The status-quo in runtime and optimizations design favors overlap and fast injection. For the systems examined in this chapter we observe a performance inversion between injection speed and throughput: the networking layers allowing the fastest injection rate observe the highest throughput degradation. We have re-implemented all of our microbenchmarks using the vendor APIs OpenIB Verbs and DMAPP. While calling the native API provides the fastest injection rate, those benchmarks achieve lower throughput than either GASNet, UPC or MPI. The detailed results are omitted for brevity. We believe that increasing the number of cores per node will require a shift towards optimizations for throughput using new approaches and performance metrics. Our congestion avoiding runtime samples points in the space of throughput oriented designs and we believe the Proxy design can provide both fast injection/overlap and throughput.

The microbenchmark results in Section 5.1 indicate that congestion is observable independently of the implementation, i.e. GASNet, MPI or native APIs, or the communication paradigm, i.e. one-sided in GASNet and two-sided in MPI. On the InfiniBand system
we have experimented with multiple NIC resource knobs controlled by software: the settings used in this study provide the best default performance. The MPI implementations (Cray MPI, MPICH, OpenMPI) seem to be affected even more than the one-sided runtimes (GASNet and Cray UPC). Thus, deploying similar mechanisms into MPI implementations is certainly worth pursuing. The implicit flow control provided by MPI Send and Receive operations allows for extensions using closed loop control techniques.

Our congestion avoidance mechanisms implement a core stateless approach where decisions are made at the edge of the network (nodes), without global state information about actual congestion in the network core (switches). The results indicate that we can provide good performance at scale, but the question remains how close to optimal we can get and whether mechanisms using global state can do better. While we do not have conclusive evidence, our conjecture is that addressing congestion at the edge of the network is likely to provide similar or better performance than global state mechanisms at scale. Another question is that of fairness when not all the nodes in the system use congestion avoiding runtimes. Our experiments were run on capacity systems and the benchmarks were competing directly against applications using unmodified runtimes. This indicates that a congestion avoiding runtime competes well with greedy traffic participants.

This work also raises the question whether the runtime can displace the algorithm. Previous work proposes application algorithmic changes that affect the communication schedule to reduce the chance of route collision. Our implementation throttles communication operations and implicitly reduces the chance of collisions. Furthermore, Proxy can be extended with node-wide message reordering and coalescing optimizations and mechanisms to avoid route collision can be provided at that level. Understanding the tradeoffs between these alternatives is certainly important and is the subject of future work.
Finally, we believe that our open loop runtime congestion avoidance mechanisms are orthogonal to the vendor provided closed loop congestion control mechanisms. In all of our experiments the vendor congestion control mechanisms (e.g. IB CCA) were enabled. However, the question remains if there are any undesired interactions between the two mechanisms.

5.5 Related Work

Congestion control in HPC systems has received a fair share of attention and networking, transport or runtime layer techniques to deal with congestion inside high speed networks have been thoroughly explored. Congestion control mechanisms are universally provided at the networking layer. For example, the IB Congestion Control mechanism [3] specified in the InfiniBand Architecture Specification 1.2.1 uses a closed loop reactive system. A switch detecting congestion sets a Forward Explicit Congestion Notification (FECN) bit that is preserved until message destination. The destination sends a backward ECN bit to the message source, which will temporarily reduce the injection rate. Dally [47] pioneered the concept of wormhole routing and his work has been since extended with congestion free routing alternatives on a very large variety of network topologies. For example, Zahavi et al. [110] recently proposed a fat-tree routing algorithm that provides a congestion-free all-to-all shift pattern for the InfiniBand static routing.

scatter and gather for large scale InfiniBand clusters. Thakur et al. [101] discussed the scalability of MPI collectives and described implementations that use multiple algorithms in order to alleviate congestion in data intensive operations such as all-to-all.

A common characteristic of all these approaches is that they target congestion in the network core (or switches): the low level mechanisms use reactive flow control while the “algorithmic” approaches use static communication schedules that avoid route collision. Systems with enough cores per node to cause NIC congestion have been deployed only very recently and we believe that our study is the first to propose solutions to this problem.

The work closest related to ours in the HPC realm has been performed for MPI implementations and mostly on single core, single threaded systems. In 1994 Brewer and Kuszmaul [30] discussed how to improve performance on the CM-5 data network by delaying MPI message sends based on the number of receives posted by other ranks. Chetlur et al. [43] proposed an active layer extension to MPI to perform dynamic message aggregation on unicore. Pham [89] also discussed MPI message aggregation heuristics on unicores and compared sender and receiver initiated schemes. These techniques use a message aggregation threshold and timeouts with a result equivalent to message rate limitation within a single thread of control. In this chapter we advocate for node wide count based message limitation and empirically compare it with rate limitation extended to multi-threaded applications. Furthermore, in MPI information about the system wide state is available to feedback loops (closed control) mechanisms by matching Send and Receive operations. Since in one-sided communication paradigms this type of flow control is not readily available, our techniques use open loop control with heuristics based only on node local knowledge.
5.6 Summary

In this chapter, we showed that contemporary networks or runtime layers are not very well equipped to deal with a large number of operations in flight and suffer from congestion. We distinguish two types of congestion: Rate Congestion happens when tasks inject too many concurrent messages, while Concurrency Congestion happens when too many cores are active at the same time. We propose a runtime design using proactive congestion avoidance techniques: a thin software layer is interposed between the application and the runtime to limit the number of concurrent operations. This approach allows the communication load to increase to the point without actually triggering native congestion control mechanisms. We implement a congestion avoiding runtime for one-sided communication on top of two UPC runtimes for two networks: InfiniBand and Cray Gemini. We discuss heuristics to limit the number of messages in flight and present implementations using either task inline or server based mechanisms. The runtime can provide performance and performance portability for all-to-all collectives, fine grained application benchmarks, as well as implementations of the NAS Parallel Benchmarks. Microbenchmark results indicate that throttling the number of messages in flight per core can provide up to 4X performance improvements, while throttling the number of active cores per node can provide additional 40% and 6X performance improvement for UPC and MPI respectively. We evaluate inline (each task makes independent decisions) and proxy (server) congestion avoidance designs. We improve all-to-all collective performance by up to 4X and provide better performance than vendor provided MPI and UPC implementations. We also demonstrate performance improvements of up to 60% in application settings.
Chapter 6: REDUCING NETWORK CONTENTION OF MULTI-THREADED UPC RUNTIME THROUGH MULTI-ENDPOINT

In previous chapters, we have focused on process-based MPI/UPC runtimes and contention problems related to them. As the number of cores in each node keeps increasing, intra-node communication has gained more and more portion in applications. As a result, thread-based runtime are being emphasized again due to its low-latency. Furthermore, thread-based runtime can provide low-level load balancing schemes by utilizing the feature of threads: shared address space. MPI+OpenMP hybrid mode is already being utilized in many HPC applications. UPC also has both process-based runtime and thread-based runtime implementation. However, with existing process-based middleware designs, threads are suffering very bad inter-node communication performance, due to the contentions from accessing network resources. From this chapter, we shift our concern from process-based runtimes to thread-based runtimes and multi-endpoint designs.

In this chapter, we first analyze various design alternatives in-depth for a high performance and scalable UPC runtime, from several aspects: performance, portability, interoperability and support for irregular parallelism. Based on the conclusions, we present a novel design of a multi-threaded communication runtime that supports multi-endpoints. We also discuss and propose two dynamic load balancing schemes: true network helper thread and
work stealing, in order to study the potential benefits of multi-endpoints design in irregular applications.

The rest of this chapter is organized as follows. In Section 6.1 we present our analysis of the design alternatives for a UPC runtime from several aspects. In Section 6.2, we discuss the details relating to the multi-threaded design and multi-endpoint. Then we propose dynamic load balancing schemes based on the new runtime in Section 6.3. In Section 6.4, we evaluate our designs and present the results. Finally, we discuss related work in Section 6.5. Finally we conclude this chapter in Section 6.6.

### 6.1 Analysis of Design Alternatives for UPC Runtimes

A UPC application consists of a set of UPC threads. A multi-threaded runtime could assign one thread to each UPC thread, whereas a non multi-threaded runtime (process based), can assign one process to each UPC thread. Figure 6.1 illustrates the two choices. Recently, Blagojevic et al. suggest running a hybrid of process and thread environments to overcome some of the locking overheads in their design [27]. However, even in this case, UPC threads still share network endpoints in the same node. The Berkeley UPC runtime offers a feature called PSHM (Inter-Process SHared Memory) [27]. This feature provides support for UPC shared arrays using SYSV shared memory. Using PSHM, even non-multi-threaded runtime can support direct read/write into UPC shared arrays.

In rest of this section, we present our analysis of the design alternatives for a UPC runtime from several aspects, such as: performance, portability, interoperability, compatibility and support for irregular parallelism. We evaluate these aspects based on several UPC runtime design alternatives: Process with PSHM, Process without PSHM, Multi-Thread with one endpoint (Global Locks), Multi-Thread with one endpoint (Fine-grained Locks), and
Figure 6.1: Non Multi-threaded and Multi-threaded choices for UPC Runtime
Multi-Thread with multi-endpoints (Lock free). These design alternatives are shown in Figure 6.2.

1. **Optimization for Irregular Applications**: The multi-threaded design based on multiple endpoints has the potential for providing the best performance for irregular applications. This is because the communication and computation state of all processor cores can be shared among the runtime threads in multi-threaded design. At the same time, multiple endpoint based runtime outperforms single endpoint based runtime by reducing contentions from network. On the other hand, sharing of processor states is either impossible or extremely complicated in the process based approaches, especially without application level interference involved. As a result, idle cores cannot touch the incoming communication requests of the overloaded cores. Therefore, some cores in the node can be completely idle when work is not evenly distributed. This results in low overall CPU utilization.

2. **Inter-node Communication**: The lack of any locking requirement helps alternative designs (based on either processes or multi-endpoints) perform better in inter-node performance. The thread lock based designs have some penalties for network communication, since there is always some degree of serialization.

3. **Intra-node Communication**: All thread based designs are good in intra-node communication performance due to address space sharing. The PSHM feature helps process based design to achieve similar intra-node performance as thread based designs. However, PSHM brings a slight portability hit, as discussed in the following portability discussion. The process-based design without PSHM will have degraded
intra-node performance since distributed UPC arrays within the node will not be directly accessible by UPC threads, requiring buffered communication through IPC.

4. **Portability:** Process based approach using PSHM requires pre-allocated shared memory size of the order of entire memory on the node since it is expected that UPC applications will use arrays that are a significant fraction of available memory. This is a fairly large amount of shared memory, and not all systems may be able to enable it. In some cases, there is a system limit on the amount of shared memory pages. On the other hand, thread based designs have no special requirement for OS, since address space sharing is a core feature of threads. Process based approach without PSHM is very portable, but has lower intra-node performance.

5. **Compatibility:** The process based case provides good compatibility with existing MPI and scientific libraries. This is very important as these libraries have hundreds of man-years invested in them, and it is very hard to re-implement them. With the thread based approaches, care needs to be taken before accessing third-party libraries as they may not be thread safe. However, this situation is currently unclear. The UPC specification [102] itself does not specify whether UPC threads are processes or threads. Therefore, the developer of a portable UPC application must consider both cases. Additionally, modern MPI libraries, such as MPICH2 [9] and MVAPICH2 [10] support full multi-threading (MPI_THREAD_MULTIPLE). Thus, we do not stress on compatibility aspects in this chapter, as the situation is unclear from the UPC specification itself.

Finally, the dotted line indicates the overall salient points of the multi-endpoint design and how it scores high on all the design alternatives.
Figure 6.2: Multi-Dimensional Comparison of Design Alternatives: Dotted line indicates coverage of design space by multi-threaded runtime with multiple network endpoints
6.2 Design Details of Multi-threaded Runtime

In this Section, we discuss the details relating to the multi-threaded design and multi-endpoint.

**Global Lock:** This is the easiest method to maintain thread safety based on existing UCR runtime library. All APIs between GASNet and UCR are protected by locks. Thus, no two threads can enter the runtime at the same time. While this option provides a simple performance model and straightforward implementation, it suffers from contention when multiple threads try to access the network at the same time.

**Fine-grained Lock:** In this design approach, we use multiple locks in the UCR runtime. Using multiple locks, multiple threads can access different parts of the runtime simultaneously. As shown in Figure 6.1(b), every critical component which cannot be shared across threads at the same time is protected by its own lock. For example, when thread 0 is busy copying message into or from communication buffers, thread 1 is able to access incoming and outgoing communications along with completion queues. Fine-grained locks are able to provide partial overlap for the utilization of network runtime between threads. Current GASNet InfiniBand conduit supports multi-threaded runtime by fine-grained lock. We also explore the fine-grained lock performance based on UCR and the results are presented in Section 6.4.2.

**Multi-Endpoint:** Any locks that are required for thread safety will add overhead for accessing network resources. The only way to achieve a lock-free design is to provide independent network endpoints and related connection context for each thread, as shown in Figures 6.3 (a) and (b). Threads are connected to their endpoints using certain mapping relationships. Thus, the threads can access the network resources without affecting others. There are two ways to realize such mapping relationships. The first one is to have totally
Figure 6.3: UCR: Thread Mapping with Multi-Endpoint

(a) UCR with additional communication thread

(b) UCR with support for work stealing
independent threads with their own thread local heaps. This will totally hide the endpoint from other threads. Other threads will not be able to “peek” into this thread’s context. The second approach is to let the parent thread (when the process is first launched) create the whole runtime structures for every thread, where the pointers to each thread’s local heap are stored. We choose the second scheme, as this preserves the possibility for further optimization techniques such as communication threads. In order to realize the second scheme, different network end-point structures are kept in an array; then `pthread_key_create` function will generate a key corresponding to every thread’s id; later on, whenever a thread wants to access its end-point, it calls `pthread_key_get` to match the right thread index for it. Thus, all APIs provided by UCR will require an additional `thread_index` in their arguments to identify the correct end-point.

6.3 Load Balancing for Irregular Applications Based on Multi-endpoint

Applications exhibiting irregular patterns of computation and communication are often hard to express. PGAS languages provide global address space, which alleviates some of the programmability challenges. However, the programmer still has to address performance concerns and keep similar overall load on processing cores. This involves programming effort and the additional logic which hurts programmer productivity. Strategies for load balancing may vary significantly among different systems. Compiler based approaches may also be used, but they may be limited by lack of dynamic characteristics. We present novel load balancing schemes which completely reside inside the runtime layer, and provide transparent optimization to irregular applications. These schemes are application independent. Thus they can provide generic load balancing, even for applications that don’t have specific application-level optimizations.
6.3.1 Dedicated Communication Threads

The advent of multi-core computing has resulted in a plethora of cores, even though memory bandwidth remains expensive. It is a common situation that many commodity High End Computing (HEC) servers have relatively more cores than memory bandwidth. Often, processing cores are busy with application computation, when new communication requests arrive. New data requests from these threads will be blocked until computations have finished. This situation happens when underlying communication uses active messages. This is because the handling of incoming active messages need the receiver’s CPU to be involved. In such situations, network bandwidth is wasted. The PGAS programming model when coupled with a multi-threaded runtime has the potential to utilize extra cores for performing communication related operations.

Using our multi-threaded and multiple endpoint design, this communication helper thread optimization can be achieved. Figure 6.3(a) illustrates this design idea. Using this design, UCR can be configured to run an extra thread that is not visible at the UPC level. This thread is used exclusively for polling network operations while other threads are busy. When the UPC threads are busy doing computation, this helper thread will enter the runtime to poll each UPC thread’s endpoint and handle incoming active messages on behalf of busy UPC threads. This design relies on the “trylock” pthread APIs. Using this, unnecessary contention can be avoided.

6.3.2 Work Stealing for Efficient Asynchronous Remote Methods

Work stealing can be natively supported by the multi-threaded runtime with multiple-endpoint design. Using asynchronous remote methods, such as those proposed in [98],
UPC application developers can rely on the underlying runtime to load balance computations on destination computation nodes. In an irregular application, different computation requests can be sent to various UPC threads on the same node. Based on how the requests are received, each core will perform the computation assigned to it. Using a multi-threaded runtime, idle cores can “steal” work belonging to other cores.

Our design of multi-endpoint runtime enables work stealing mechanism to balance workload between UPC threads on the same node. Using work stealing, UPC threads that are idle will access the endpoints of threads that are busy in computation. This is shown in Figure 6.3(b). When incoming requests are for data transfer, the UPC thread performing work stealing can directly reply back to the requestor. When the incoming request is requiring for computation, any idle UPC thread can pick up the computation request and start executing it. This is demonstrated in Figure 6.9(d). This is possible as in multi-threaded implementation, threads share the same space thus there is no overhead to transplant a computation work from one thread to another local thread.

We evaluate the possible benefits that can be achieved by communication helper threads and work stealing in Section 6.4.4-6.4.6.

### 6.4 Performance Evaluation

In this section, we present our performance evaluation of the different design alternatives mentioned in Sections 6.1 and 6.2. Performance evaluation includes microbenchmark level evaluations of UPC memput/memget calls and message rate analysis. We also introduce two novel benchmarks to measure performance for irregular communication and computation. We chose the following alternatives for microbenchmark evaluation: 1) ibv-process: the process based implementation from Berkeley UPC GASNet
IBV-conduit; 2) ibv-thread: the multi-threaded with single endpoint implementation from Berkeley UPC GASNet IBV-conduit, which uses fine-grained locks introduced in Section 6.2; 3) UCR-global-lock: Multi-threaded implementation based on single endpoint UCR codebase, which uses global locks to achieve thread safety; 4) UCR-fg-lock: Multi-threaded implementation based on single endpoint UCR codebase, which uses fine-grained locks; 5) UCR-endpoint: Multi-threaded UCR library based on multiple endpoints.

6.4.1 Experimental Platform

The experiments were carried out on Intel Xeon cluster, equipped with Mellanox ConnectX QDR HCAs (32 Gbps data rate). The entire cluster consists of 1,280 cores where each node has eight Intel Xeon EE5630 processors, organized into two sockets of four cores each clocked at 2.53 GHz. The L1 cache is 32K, L2 is 256 K and shared L3 (among cores in one socket) is 12 M. Each node has 12 GB of main memory. Red Hat Enterprise Linux Server Release 5.4 was used on all nodes with OFED version 1.5.1. Berkeley UPC version 2.12.1 with PSHM (sysv) enabled is used for the following evaluations.

6.4.2 Microbenchmark Level Evaluation

We chose three representative benchmarks for the microbenchmark level performance analysis: `upc_memput`, `upc_memget` and the asynchronous versions of `upc_memput` (non-blocking `upc_memput`). The `upc_memput` call writes specified amounts of data bytes to the remote side. Similarly, `upc_memget` fetches specified amounts of data bytes from the remote side. The non-blocking versions of these calls return immediately, without waiting for the transfer to be complete. These calls return a handle, which can be used later to wait or detect completion. The asynchronous (non-blocking) version of `upc_memput` is an
Figure 6.4: Small message latency performance

Figure 6.5: Large message latency performance

Figure 6.6: Bandwidth performance
extension of the Berkeley UPC runtime, which is not defined as a standard function in UPC Language Specifications [102].

We present the bandwidth and latency results for message size up to 2 MB. All the data averaged over 1,000 samples. The performance evaluation presented in this section corresponds to inter-node communication, with two nodes fully occupied by 16 UPC threads. The 16 UPC threads are grouped into eight pairs and corresponding operations happen between each pair of UPC threads. Since all the UPC threads in one process can directly access memory within the process, it does not make sense to compare the intra-node communication performance.

Figures 6.4(a), 6.5(a), and 6.6(a) depict the performance results of the design alternatives for upc_memput. We observe that UCR-global-lock performs the worst. This is due to contention between the threads to acquire the global lock for accessing the network resources. Each thread will wait to acquire the global lock before beginning its operation. The UCR-fg-lock and ibv-thread schemes are based on fine grained locks. They can significantly improve latency for small messages by overlapping buffer management, posting send requests and polling completion queues. There are different locks for each of these operations. Even though UCR-fg-lock and ibv-thread both use fine grained locks, UCR-fg-lock is able to lower latency by 3 $\mu$s and increase bandwidth by 35 MB/s compared to ibv-thread. This is because in the UCR-fg-lock design, completion queue is protected by mutex_trylock instead of mutex_lock. This ensures more flexibility for threads to avoid contention when multiple threads are competing for the same lock. The same pattern can also be observed for two other operations: upc_memget in Figures 6.4(b), 6.5(b), and 6.6(b); upc_memput_async in Figures 6.4(c), 6.5(c), and 6.6(c).
The data reveals that multi-threaded methods which are based on a single endpoint suffer from overhead as compared to process based approaches. The multi-endpoint design UCR-endpoint is able to successfully reduce latency further by more than 80%. This is the same latency seen by process based schemes, as shown in Figures 6.4(a), 6.4(b) and 6.4(c). This is due to lock-free multiple endpoint design. Threads can enter UCR runtime in parallel for all components, similar to that of the processes.

6.4.3 Message Rate Evaluation

![Graphs showing message rate evaluation](image)

(a) Message Rate (8-pairs) comparing various design alternatives
(b) Multi-pair Message Rate with UCR-endpoint

Figure 6.7: Message Rate Performance

We developed a UPC message rate benchmark, which evaluates the aggregate unidirectional message rate between multiple pairs of UPC execution threads. Each of the sending UPC-thread sends a fixed number of messages (window size) back-to-back to the paired receiving UPC-thread. It then waits for an acknowledgment from the remote side.
This process is repeated for several iterations. The objective of this benchmark is to determine the achieved message rate from one node to another node.

This benchmark executes 16 UPC-threads grouped into eight pairs. Each group of eight threads runs on a node, fully occupying all the cores. This is the same as micro-benchmarks described in Section 6.4.2. The results of this benchmark are presented in Figure 6.7(a). Measurements are averaged over 1,000 samples.

We observe from the results that UCR-global-lock, UCR-fg-lock and ibv-thread do not achieve the maximum possible message rate. In fact, the message rates of multi-thread designs with one endpoint are almost one-eighth as compared to ibv-procs or UCR-endpoint. This is because, there is only one network endpoint. Whereas in ibv-procs or UCR-endpoint, there are multiple (eight in this experiment) network endpoints. UPC-threads can freely access their own endpoints. As the message size increases, the message rate in all designs will converge. This is the point where the network bandwidth is saturated. However, for smaller messages, having more endpoints exploits concurrency in the network adapter.

We also present the evaluation results of UCR-endpoint by gradually increasing the number of pairs of communicating UPC threads. These results are shown in Figure 6.7(b). We observe that message rates for smaller messages is dependent on the number of pairs. The message rate of two pairs is double that of one pair. Correspondingly, the message rate of four pairs is double that of two pairs. With eight pairs, however, we see that the message rate doesn’t increase any more. This is due to the fact that most concurrency in the network adapter is already utilized with four pairs. Due to increasing number of endpoints, the adapter needs to look up more endpoints. We note that even the eight pair message rate
with eight endpoints is almost four times that of the one endpoint, which justifies the use of multiple endpoints in the UPC runtime.

### 6.4.4 Dedicated Communication Thread - Bandwidth Evaluation

The UCR multi-threaded design can be configured to run a dedicated communication thread. This is indicated in Section 6.3.1. In this design, the communication thread is responsible for handling incoming and outgoing network requests. Other UPC threads can be busy in computation. We developed a GASNet-level benchmark, which mimics the characteristics of an irregular application, in which some UPC-threads are doing busy computation and cannot contribute to communication progress. This benchmark demonstrates how the dedicated communication thread design can benefit such applications.

Two nodes with seven UPC-threads running on each are used for this benchmark. The 14 UPC-threads are grouped into seven pairs, with one sender and one receiver. The senders send 1 MB sized messages to their peers and then wait for acknowledgement. The receivers perform a defined amount of computation before they are able to poll for incoming messages and send reply back. Without communication helper thread, the original senders can only get replies after their peers finished the computation. However, when communication helper thread is enabled (it can reside on the idle CPU), the messages can be discovered by the helper thread and are processed immediately. We use GASNet level active messages (long async) for this benchmark. The benchmark reports the bidirectional bandwidth, with increasing amounts of computation on the receiver side. Results of this benchmark are shown in Figure 6.8(a).

The results indicate that in the case of ibv-process and UCR without helper thread, the increase in computation on the receiver side drastically affects the observed communication
bandwidth. This is due to the communication link remaining idle while UPC threads are busy computing, as shown in Figure 6.8(b). In the case of UCR multi-endpoint design with communication thread, bandwidth is not affected. This is due to the thread taking care of communication requests while other UPC threads are busy computing.

### 6.4.5 Work Stealing - Communication Evaluation

The work-stealing design of UCR multi-endpoint is described in detail in Section 6.3.2. To evaluate the possible performance improvements that can be achieved with this concept, we developed two GASNet level benchmarks, which mimic the communication and computation characteristics of irregular applications. These benchmarks are intended to simulate the *asynchronous remote method* requirement for irregular parallel applications. Each benchmark involves two nodes, each one fully occupied by eight UPC threads. The first benchmark deals with communication performance, and the second deals with computation. Evaluation with the first benchmark is presented in this section, whereas the computation benchmark is presented in the following section.

The first benchmark is communication oriented. It is similar to the one mentioned in Section 6.4.4, in which sender UPC-threads in node 1 send multiple fixed size (1MB) data requests to the peer UPC-threads on node 2. However, in this benchmark, receiver UPC-threads will do different amounts of computations. Bandwidth is measured on the sender side, based on the time senders have to wait for the reply from receivers. The difference between communication helper thread and communication work stealing design is that, in the latter, all threads are involved in computation. Whenever one UPC-thread becomes idle, it helps in the communication progress of other UPC-threads on the same node. Whereas in the case of the communication helper thread, one user invisible thread is dedicated for
Figure 6.8: Dedicated Communication Thread Design - Bandwidth Evaluation

(a) Bandwidth Results with Computation on Receiver Side
(b) Without UCR Dedicated Communication Thread
(c) With Dedicated Communication Thread

Figure 6.9: Work-Stealing Evaluation with unbalanced computation requests

(a) Bidirectional Bandwidth with computation on receiver side
(b) GFLOPS performance with irregular workload on receiver side
(c) Without Work Stealing Design
(d) With Work Stealing Design
helping in the communication progress of all UPC-threads busy computing on the same node.

Figure 6.9(a) depicts the bidirectional bandwidth measured at the sender side, with increasing computation ranges on the receiver side. The bandwidth reduces with increasing computation because throughput is limited by the least loaded UPC-thread. The work stealing version performs to the best possible (bounded by least loaded UPC-thread). We can observe that the ‘UCR-endpoint with work stealing’ performs the best for all ranges of computation. It achieves peak bandwidth of 2,750 MB/s when no computation is present and keeps 1,750 MB/s bandwidth when the computation range between receiver UPC-threads increases. But for other designs (without work stealing), the bandwidth drops severely as the computation range increases between receiver UPC-threads. For a computation range of 0.2 seconds to 11.4 seconds, the bandwidth is just around 500 MB/s.

### 6.4.6 Work Stealing - Computation Evaluation

We have developed two benchmarks to simulate irregular applications using asynchronous remote methods. The first benchmark relating to communication was presented in the previous section. In this section, we present the second benchmark which deals with computation. Computation is represented by DGEMM, a matrix-matrix multiplication program. Sender UPC-threads send varying workload requests (in terms of matrix size) to respective receiver UPC-threads. Receivers reply back once they finish corresponding computations and Floating Point Operations per Seconds (FLOPS) are measured, based on the total computations involved and time used by the whole benchmark. We compare the results with the estimated theoretical peak of the system, based on the CPU clock frequency (2.53 GHz) and the fact that the Xeon processors used in our testbed have four
floating point units (10 GFLOPS per core). We also measured the maximum achievable GFLOPS by having each thread do purely local DGEMM 2000 matrix-matrix multiplication with all cores on the node fully occupied, which gives a result of 9.5 GFLOPS per core. Without work stealing, receivers with light workload become idle after finishing their own computation workload while receivers with heavy workloads keep busy all the time with a large amount of computation (Figure 6.9(c)). Thus with more unbalanced workload on the receivers, more CPU cycles are wasted, which results in a poor GFLOPS performance. However, with work stealing supported by multi-endpoints design, idle threads consume the workload requests posted to the busy threads as their own work and thus balance the workload on the receiver node (Figure 6.9(d)).

Figure 6.9(b) demonstrates the GFLOPS performance of this benchmark. We define 2,000 matrix size as the average workload. As the x axis increases from 0 to 2,000, on the receiver side, three UPC-threads will receive workload requests of size = (2,000 -x); another three UPC-threads will receive workload requests of size = (2,000 + x); the left two UPC-threads will keep getting requests of size = 2,000. As x increases, the workload requests on different UPC-threads become more and more unbalanced.

With x = 0, all the threads are equally balanced. All three schemes can reach the maximum achievable peak value: 76 GFLOPS. However, as long as the threads are loaded differently, the designs without work stealing will suffer. This is due to some cores becoming idle and the others becoming heavily loaded. Thus, the overall GFLOPS rating will drop.

Using ‘UCR-endpoint w/ work stealing’ all cores are dynamically balanced for the whole benchmark execution. Thus, the GFLOPS value is almost the same as the maximum
achievable value no matter how the initial workload is distributed. The work stealing design helps UCR-endpoint achieve 1.3 times better performance than the other two situations when the difference range reaches 2,000. This illustrates the fact that work stealing concepts can be applied to balance workloads in irregular applications and achieve comparable performance as that of applications with regular communication/computation patterns.

6.5 Related Work

There are several implementations of the UPC language. The Berkeley UPC implementation [72] is a popular open-source implementation of UPC. IBM and Cray also distribute their own versions of UPC implementations specifically optimized for their platforms. Barton et. al. have proven the scalability of the IBM UPC implementation on the Blue Gene architecture [35]. In this chapter, we focus on the Berkeley UPC implementation.

Duell studied the question of mapping UPC threads to OS processes or threads in detail in [54]. Duell concluded that having multiple network endpoints for a threaded PGAS runtime would result in more resource consumption. The main reason for this conclusion was that \(O(n^2)\) buffers were required for every endpoint. Duell also highlights the fact that sharing one network endpoint could have the detrimental impact of lock contention for each network access. Blagojevic et al. have recently studied this issue as well [27]. Their evaluations show that single network endpoint access is the bottleneck for network latencies and throughput. They have suggested using a hybrid of processes and threads within the runtime. Neither Duell nor Blagojevic have presented an evaluation of a multi-endpoint based design.
In our work, we demonstrate a multi-network endpoint based design for multi-threaded UPC runtime. We utilize advanced network features that can eliminate contention of multiple threads for network access, while keeping memory requirement low. We address the memory consumption issue through InfiniBand’s Shared Receive Queues [100] and Unreliable Datagram [68].

Irregular parallelism has always been a challenge for application scientists. Some PGAS languages, such as X10 [39] have attempted to address this by having language-level support for work stealing techniques. Cilk [99] is another language which supports work stealing. Cilk provides a set of productive, and easy to use concurrency primitives. Applications exhibiting irregular computation and communication patterns often are unable to predict the amount of work to be done at any stage of the algorithm. In such environments, application developers prefer to have asynchronous remote methods. This is exemplified in the work presented by Shet et. al. [98]. One of the key challenges of achieving high performance from asynchronous remote methods is to make sure all cores are relatively load balanced.

We note that such load balancing among cores is much easier when the runtime is multi-threaded. Cilk, for example, is based on a multi-threaded runtime. Sharing and stealing work in a process based runtime is extremely hard to accomplish. In that context, our work with a multi-endpoint multi-threaded runtime is ideally suited to not only offer excellent network performance, but also enable support for load-balancing by work stealing with lower overhead.
6.6 Summary

In this chapter, we explored multiple design alternatives and the overall design space for UPC runtime implementation on multi-core architectures. Based on the evaluation and comparison between OS-process mapping and OS-thread mapping schemes, we investigated and presented a new multi-threaded runtime with multiple network endpoint design in Unified Communication Runtime (UCR). The new multi-threaded UCR-endpoint design is not only an improved version of UCR to enable multi-threading, but it also provides significantly better performance than existing GASNet multi-threaded runtime. Our runtime is able to dramatically decrease network access contention resulting in 80% lower latency for fine-grained memget/memput operations and almost doubling the bandwidth for medium size messages, compared to multi-threaded Berkeley UPC Runtime. We also presented novel benchmarks that attempt to capture communication and computation patterns of irregular applications. The results from these novel benchmarks demonstrate that our design is able to provide significant improvement to these irregular benchmarks with the aid of ‘communication helper threads’ and ‘work stealing’ concepts. These schemes provide runtime-level load balancing, which can only be utilized by thread-based runtime on multi-core architectures. Our evaluation with novel benchmarks shows that our runtime with proposed load balancing schemes can achieve 90% of the peak efficiency, which is a factor of 1.3 times better than existing Berkeley UPC Runtime.
Chapter 7: HIGH PERFORMANCE LOCK-FREE MULTIPLENDPOINT RUNTIME FOR MPI/OPENMP HYBRID APPLICATIONS

Parallel applications are being increasingly developed based on the MPI/OpenMP hybrid model. MPI/OpenMP applications typically utilize the “funneled” or the “single” mode that allows only one thread to perform MPI operations; and, this leads to inefficient resource utilization. Further, state-of-the-art MPI libraries rely on locking mechanisms to guarantee thread-safety and this discourages application developers from using multiple threads to perform MPI operations. In last chapter, we addressed the importance of multi-endpoint design for thread-based UPC runtime. However, the contention issue and locking overhead exist not only in UPC, but also in the MPI/OpenMP hybrid model. Thus, we continue our study of multi-endpoint and multi-threaded runtime for MPI/OpenMP environment. Furthermore, different from UPC programming model, another key problem in MPI/OpenMP hybrid model is how to provide the benefits of multi-endpoint to existing applications with minimal efforts from users.

In this chapter, we address many important challenges associated with designing a multi-threaded MPI runtime that offers high performance lock-free communication for MPI/OpenMP applications, based on the multi-endpoint design. Our designs can significantly improve the latency of point-to-point and collective operations. We also propose a
transparent-control mode to enable parallel MPI/OpenMP applications to directly leverage our designs.

The rest of this chapter is organized as follow: We first identify the fundamental problems limiting the communication performance of current MPI/OpenMP hybrid applications in Section 7.1. Then we discuss the various design alternatives of our approach in Section 7.2. In Sections 7.3 and 7.4, we discuss how to deliver the benefits of Multi-Endpoint runtime to applications, for point-to-point and collective operations, respectively. The associated performance benefits are discussed in Section 7.5. We discuss another implementation scheme “helper-thread” mode and memory footprint of our proposed design in Section 7.6. We discuss related work in Section 7.7. Finally we conclude this chapter in Section 7.8.

7.1 Motivation

On emerging multi-/many-core architectures, the MPI/OpenMP hybrid paradigm is widely believed to improve the performance and scalability of parallel applications. However, previous studies have demonstrated that several critical issues need to be carefully considered to extract the maximum performance benefits of such a hybrid model [20, 24, 29, 36, 71]. In this chapter, we explore the following important issues: i) interaction between the MPI and the OpenMP interfaces and ii) communication performance offered by the underlying MPI runtime for hybrid MPI/OpenMP applications. In this section, we specifically draw out the potential performance pitfalls of the state-of-the-art support for MPI/OpenMP hybrid models.

Master-Only Model: Many current generation parallel applications that are based on the hybrid MPI/OpenMP paradigm rely on the Master-Only model. In this approach, only the
master thread performs MPI communication operations, while the rest of the threads are idle for the duration of the communication phases. On emerging multi-/many-core architectures, this leads to poor resource utilization and higher communication and synchronization overheads. This is primarily because one thread alone may not always completely saturate the network bandwidth, even with the latest InfiniBand interconnect.

**Multi-Threaded Support in MPI runtimes:** Hybrid applications can be re-designed to utilize dedicated threads to perform communication operations while the remaining OpenMP threads help accelerate computation. However, such a re-design is not always straight-forward [94] because the applications should also re-distribute the data across the compute threads. Further, the communication performance of such applications is tightly coupled with the features offered by the underlying MPI implementations, specifically for multi-threaded applications. The MPI specification mandates that implementations should guarantee thread-safety; but, the choice of implementing thread safety and efficient communication progress is specific to the implementation. For example, naive MPI implementations may use a global-lock to guarantee thread-safety. However, this significantly limits the concurrency across the communicating threads and leads to poor communication performance for multi-threaded applications. Recently, MPI libraries, such as MPICH2 [9], have introduced support for fine-grained locking mechanisms to improve the performance of multi-threaded applications. However, as authors note in [24], some of the critical operations within the MPI library still need to be serialized. Further, the communication threads are also prone to CPU resource contention, locking, and scheduling overheads. Owing to these factors, multi-threaded MPI/OpenMP applications may not be able to achieve the best communication performance even with the state-of-the-art MPI implementations.
Figure 7.1: Point-to-Point communication characteristics with varying number of cores per node

**Network utilization:** Depending on the communication patterns exhibited by the parallel application, one communicating thread may not always fully utilize the available network bandwidth. In Figure 7.1 we report our observations with the `osu_mbw_mr` benchmark that is available in the OSU Micro-Benchmark (OMB) suite [14]. We first measure the bandwidth achieved with one pair of processes, executing on two different compute nodes. Next, we measure the bandwidth by varying the number of pairs and report the percentage difference in the bandwidth against the one-pair case. We note that for small and medium length messages, we observe better bandwidth as we increase the number of active cores per node. However, for very large message lengths, we note that one process per node can nearly saturate the network. Hence, it is necessary for communication libraries to determine the optimal way for scheduling the data transfers to minimize network congestion and maximize the utilization of the network bandwidth and available CPU cycles.
Based on these observations, we propose a high performance, lock-free Multi-Endpoint MPI runtime to specifically address the challenges of improving communication performance of hybrid applications. In Figure 7.2, we discuss the execution model of hybrid MPI/OpenMP applications in the Multi-Endpoint MPI runtime framework. Hybrid applications may choose to perform MPI collective operations in the master-only mode, or utilize dedicated threads to perform point-to-point operations. Our proposed Multi-Endpoint runtime can efficiently improve the communication performance for both of these scenarios.

![Figure 7.2: Overview of the Multi-Endpoint framework](image)

Figure 7.2: Overview of the Multi-Endpoint framework
7.2 Designing Multi-endpoint Communication Runtime

7.2.1 Lock-free Communication Runtime Based on Multi-endpoint

In Section 7.1, we discussed the performance characteristics of the state-of-the-art MPI libraries for hybrid multi-threaded applications. We note that current generation MPI libraries guarantee thread-safety through fine-grained locking mechanisms and there are opportunities for further improving the communication performance of multi-threaded applications. By breaking down the existing MPI runtime and identifying the critical components in thread-aware manner, we propose to create a MPI runtime with lock-free communication routines for ensuring high performance for multi-threaded applications.

![Diagram](image)

**Figure 7.3:** Components of the multi-endpoint based lock-free communication runtime

We discuss the structure of the new runtime in Figure 7.3. Blue sections are the ADI3/CH3 layer. ADI3 (the third generation of the Abstract Device Interface) is a full-featured abstract device interface to provide a portability layer in MPICH2 [9]. The CH3 layer implements the ADI3 functions, and provides an interface consisting of only a dozen
functions. CH3 layer consists of several channels for different communication architectures, including the RDMA channel for InfiniBand interconnects. We identify three critical components (marked yellow in Figure 7.3), which need to be re-designed according to multi-threading requirement, based on the CH3-RDMA channel of original MPI runtime implementation:

**Request Handling:** Instead of sharing the same request memory region across different threads through locking mechanisms, our proposed design pre-allocates request memory objects according to the number of threads that would be able to call MPI functions. Each thread is associated with a thread-ID as a thread local storage parameter (LST). Any request-related operations, including create/enqueue/dequeue/destroy, should access the corresponding queues or request memory object according to a specific thread-ID. Hence, our design removes the requirements of locking the request queue for ensuring thread-safety.

**Progress Engine:** The Progress Engine is a key component for MPI runtime to handle the message transfer from network channels. For InfiniBand-based RDMA channel, the Progress Engine includes operations to access Send/Recv Queue Pair (QPs), Completion Queue (CQ), Local Keys and Remote Keys for RDMA operations, which needs to be thread-safe. In order to enable threads to launch network send/receive operations without lock protection, the new design establishes CQs and QPs according to the number of endpoints, $N$. However, in the process-based design, QPs need to be established between all communicating remote peers in the Reliable Connection mode. In our proposed multi-endpoint design, we establish connections only between “matching” endpoints. For example, the $i$th endpoint of Node 0 is only connected with the $i$th endpoints on other nodes.
Through such a design, we expect the number of connections to grow as a function of $O(N)$ instead of $O(N^2)$.

**Communication Resources Management:** Communication resources include the temporary send/receive buffers, buffers for low-latency communications, RDMA credits, multi-rail information and all other local information required for communication. These data structures, along with the new Progress Engine, are stored inside Virtual Connection (VC), in the form of a table. The Endpoint-ID serves as the index key for accessing the table. We refer a pair of Progress Engines and communication resources as a set of endpoints in the following part of this chapter. Access to the communication resources would require a specific Endpoint-ID.

When a thread needs to do communication, it first matches the LST thread-ID with its Endpoint-ID. Then this thread can access the communication resources and Progress Engine associated with this Endpoint-ID. In the reverse way, when a message is discovered from a Progress Engine, it can only be matched or posted into a certain posted/unexpected receive queue, by matching the Endpoint-ID with the thread-ID. By creating thread specific critical components, messages sent/received by different threads are processed independently with each other and thus eliminating the need for locks from these critical components of the MPI runtime.

### 7.2.2 Multi-endpoint Impact on Network Performance

In Section 7.2.1, we introduced a new MPI runtime with lock-free communication routines based on multi-endpoint design. By removing the locks from three critical components, OpenMP threads are able to make MPI communication calls as efficiently as processes. However, different from processes, thread-based endpoint communication runtime
Figure 7.4: Comparison of the Communication Overlapping in Process-based Runtime and Multi-endpoint Runtime

faces several other issues, which include: context switch; user buffer memory affinity; overhead from threads join/fork. On the other hand, thread-based multi-endpoint design also has its advantage when compared with processes, where the runtime has the flexibility to schedule different endpoints for further performance improvement. In this section, we examine the details of both runtime-level and network-level communication progress and identify how the multi-endpoint design can improve network utilization and impact communication performance.

In order to understand the underlying processing of the send and receive communication, we demonstrate the procedure as shown in Figure 7.4. For process-based design space (Figure 7.4(a)), when multiple outgoing send requests are posted by an application in a back-to-back manner, all the procedures for request handling, data copy from user buffers
to send buffers and the hand-shake protocol will be performed sequentially. The total latency for executing the set of communication requests is $n \times T_{\text{runtimeToSend}} + T_{\text{send}} + T_{\text{recv}} + n \times T_{\text{runtimeToRecv}}$ ($n$ refers to the number of in-flight requests in a single window frame); Figure 7.4(b) shows how multi-endpoint design handles $n$ messages simultaneously. The total latency now becomes: $T_{\text{runtimeToSend}} + T_{\text{send}} + T_{\text{recv}} + T_w + T_{\text{runtimeToRecv}}$ ($T_w$ refers to the extra time due to multi-endpoints). When we increase the number of endpoints involved into the communication, we reduce the latency due to parallel runtime processing. Furthermore, as different threads finish their requests at different time, the overhead of serving communication requests within the library and network communication can further overlap with each other. However, with increasing number of endpoints and concurrent in-flight requests, the $T_w$ also increases, due to memory affinity, context switching and congestion at NIC. Hence, it is necessary to arrange the endpoints according to the communication pattern, in order to achieve the best performance through the multi-endpoint runtime. We evaluate the multi-endpoint performance with different number of active endpoints, as well as, different number of concurrent in-flight requests. The Endpoint Controller is then configured according to the evaluation results, similarly as the comparison in Section 7.5.3. We also consider these factors while designing the Endpoint Controller module for point-to-point and collective schemes to provide multi-endpoint benefits to applications, as discussed in the following sections.

### 7.3 Multi-Endpoint support for Point-to-Point operations

We introduced the Multi-Endpoint runtime with lock-free communication path in Section 7.2. However, it is not straight-forward to deliver the benefits of Multi-Endpoint runtime to applications since this may also be modifying existing MPI/OpenMP applications.
Application developers have to explicitly distribute communication for OpenMP threads and this limits programmer productivity. Hence, it is necessary to explore design alternatives to provide the benefits of the proposed Multi-Endpoint runtime to real applications, in a transparent manner, or with minimal modifications. We discuss such optimizations for point-to-point communication functions and we propose a transparent support for collective operations in the next section.

$\texttt{#pragma omp parallel}
\{
  ... /*computation*/
\}

/* start communication after OpenMP threads join or use master-only model */
\texttt{MPI\_Irecv();
MPI\_Isend();
MPI\_Waitall();
$\texttt{#pragma omp parallel}
\{
  ...}

(a) Default Non-Blocking MPI Communication in Hybrid Programs

$\texttt{#pragma omp parallel}
\{
  ... /*computation*/
\}

$\texttt{#pragma omp parallel}
\{
  /* OpenMP threads call MPI functions */
  MPI\_Irecv();
  MPI\_Isend();
  MPI\_Waitall();
\}

$\texttt{#pragma omp parallel}
\{
  ...

(b) Adding only an OpenMP pragma for the new design

Figure 7.5: Minimal modifications to expose multi-endpoint for point-to-point operations

As shown in Figure 7.5(a), for the master-only model, MPI communication, non-blocking, send/receive operations are called after the OpenMP threads joined, or only the master thread makes MPI function calls through “pragma omp master” directive. Simple modifications are required for such programs in order to take advantage of the proposed Multi-Endpoint runtime. As shown in Figure 7.5(b), the application developers only need to add
an OpenMP pragma outside of the MPI functions with the same input parameters as before. Inside the runtime, the Endpoint Controller (introduced in Section 7.2.2) decides how many endpoints should be involved in executing the communication requests, according to the pre-defined tables. Next, the communication runtime divides the original communication data across $n$ available endpoints and schedules them independently. Threads that are not assigned a communication task just pass by and wait at the barrier or continue with the next unrelated communication request.

### 7.4 Transparent Re-design for Collectives Operation

In the previous section, we proposed our approach to enable hybrid applications directly leverage the benefits of our proposed Multi-Endpoint design with minor modifications. In this section, we discuss that our approach can lead to lower communication overheads for collective operations, in a transparent manner without requiring any changes to applications. In the multi-endpoint runtime, OpenMP threads are invoked inside the runtime and can concurrently perform various communication operations. We propose to re-design the collective algorithms in the MVAPICH2 library to take advantage of the available endpoints to achieve lower communication latency. In this section, we discuss such optimizations for two important collective operations: MPI_Alltoallv and MPI_Allgather.

#### 7.4.1 Alltoallv

The MVAPICH2 library [10] utilizes pair-wise exchange algorithm to implement the MPI_Alltoallv operation. If the Alltoallv operation is performed with $N$ processes, the pair-wise exchange algorithm will involve $N \times N$ send/receive operations between every pair of MPI processes. In the original single-endpoint runtime, all the send/receive requests are launched and handled by a single process. However, with our multi-endpoint design, we
note that our communication runtime can further optimize the pair-wise exchange algorithm by distributing the communication requests among a set of endpoints.

Input : num_endpoint, comm_size, my_rank  
Output: Alltoallv Communication with Multiple Endpoint  
#pragma omp parallel num_threads(num_endpoint) default(shared)  
ranks_per_group = comm_size/num_threads  
group_id = my_rank/ranks_per_group  
thread_id = omp_get_thread_num  

if thread_id == group_id then  
  for src = 0…comm_size do  
    non-blocking receive from src  
  end  
end  

for dst = thread_id * ranks_per_group … (thread_id + 1) * ranks_per_group do  
  if dst == my_rank then  
    do local copy  
  else  
    non-blocking send corresponding data to dst  
  end  
end  

complete all in-flight requests  

Algorithm 7.1: Group Pair-wise Algorithm for Alltoallv based on Multi-Endpoint Design  

In the new algorithm, all the N MPI ranks are divided into n groups, where n is equal to the number of active endpoints. In each group, the thread with Endpoint-ID equal to Group ID operates as the “receiver” (marked as yellow in Figure 7.6); and, the thread with Endpoint-ID x performs as “sender” to all the destinations in Group x. As introduced in Section 7.2.1, endpoints are connected to endpoints with same Endpoint-ID on the remote node. Thus it is necessary to make sure that the corresponding receive requests are posted by a thread with the same Endpoint-ID that is guaranteed in the new “Group” algorithm.  

105
(described in Algorithm 7.1). Furthermore, the new “Group” algorithm successfully distributes the send requests from a single endpoint across multiple endpoints. Similar to the point-to-point optimization, the runtime also relies on the Endpoint Controller to decide the number of active endpoints according to the message size and the scale of the communicator. Since our Multi-Endpoint design can perform multiple send/recv operations of the pair-wise exchange operation concurrently, we expect the potential benefits of our design to be directly proportional to the number of end-points used by the runtime. We also expect that our proposed design can lead to significant benefits for Alltoallv operations with small and medium message lengths.

Figure 7.6: Alltoallv Algorithm based on Multi-Endpoint
7.4.2 Allgatherv

In MVAPICH2, the MPI_Allgatherv operation is implemented through recursive-doubling, Bruck algorithm [31], and ring-based algorithms. The library picks one of these implementations, depending on the message size and the number of processes in the communicator. The ring-based algorithm is usually adopted for large message lengths. In this section, we discuss our approach to improve the communication latency of the ring exchange algorithm, based on our Multi-Endpoint design. The default ring exchange implements the communication operations along one-direction. In each iteration, MPI rank $i$ receives from rank $i - 1$ and sends to rank $i + 1$ and the algorithm involves $N$ iterations, with $N$ processes. This algorithm can be modified to implement the exchanges along two directions, concurrently. In this design, each MPI rank performs one set of send-recv operation with both the left and right neighbors, hence implementing the entire algorithm in $(N/2 + 1)$ iterations (described in details in Algorithm 7.2). We describe this design in Figure 7.7. However, the true benefits of this design will depend on whether the underlying network and memory sub-systems can implement the communication operations in the most efficient manner.
**Input**: comm_size, my_rank

**Output**: Allgatherv Communication with Multiple Endpoint

```c
#pragma omp parallel num_threads(2) default(shared)
thread_id = omp_get_thread_num

if thread_id == 0 then
    the first endpoint handles the clock-wise half ring
    left = (comm_size + my_rank - 1) % comm_size
    right = (my_rank + 1) % comm_size
    for i = comm_size/2...comm_size do
        sum up the amount of data to send/receive in the clock-wise half ring
        total_count += recvcounts[my_rank + i] % comm_size
    end

torecv = total_count
tosend = total_count + recvcounts[my_rank] - recvcounts[(my_rank + comm_size/2) % comm_size]

else
    the second endpoint handles the counter-clock-wise half ring
    right = (comm_size + my_rank - 1) % comm_size;
    left = (my_rank + 1) % comm_size
    for i = 1...comm_size/2 do
        sum up the amount of data to send/receive in the counter-clock-wise half ring
        total_count += recvcounts[my_rank + i] % comm_size
    end

torecv = total_count
tosend = total_count + recvcounts[my_rank] - recvcounts[(my_rank + comm_size/2 - 1) % comm_size]
end
```

(to be continued)

**Algorithm 7.2**: Double-Ring Algorithm for Allgatherv based on Multi-Endpoint (Part 1)
(Cont.)

\[ sindex = my_{\text{rank}} \]
\[ rindex = left \]

\textbf{while} tosend not equal to 0 —— torecv not equal to 0 \textbf{do}

receive from \textit{left} for \textit{sendnow} data size belong to \textit{rindex}

send to \textit{right} for \textit{recvnow} data size belong to \textit{sindex}

\[ tosend- = sendnow \]
\[ torecv- = recvnow \]

\textbf{if} \text{thread}_{\text{id}} == 0 \textbf{then}

shift the data index for next step

\[ sindex = (sindex + \text{comm}_{\text{size}} - 1) \% \text{comm}_{\text{size}} \]
\[ rindex = (rindex + \text{comm}_{\text{size}} - 1) \% \text{comm}_{\text{size}} \]

\textbf{else}

\[ sindex = (sindex + 1) \% \text{comm}_{\text{size}} \]
\[ rindex = (rindex + 1) \% \text{comm}_{\text{size}} \]

\textbf{end}

\textbf{end}

\textbf{Algorithm 7.3:} Double-Ring Algorithm for Allgatherv based on Multi-Endpoint (Part 2)

A Process-based runtime can also adopt a double-ring algorithm by issuing a non-blocking send/receive from both directions. The traffic of the second ring shares the same endpoint in this situation, that requires wait operations for two non-blocking send/receive operations at the same step. In the multi-endpoint runtime, the two rings can be completely isolated from each other, where the network/runtime procedure can concurrently execute the different communication operations. Since we are halving the number of iterations involved on the ring-exchange, we expect this design to improve the communication latency by a factor of 2, for medium message lengths. For very large message lengths, since one end-point alone can saturate the network bandwidth, we expect the benefits to be lower for such cases.
7.5 Experimental Results

In this section, we describe the experimental setup, provide the results of our experiments, and give an in-depth analysis of these results.

![Multi-Pair latency performance across different runtimes](image)

Figure 7.8: Multi-Pair latency performance across different runtimes

7.5.1 Experimental Setup

We implement the new design based on MVAPICH2 1.9a2 [10] and use it for the comparison. We carry out the micro-benchmark and application evaluations on the “Stampede” computing system from Texas Advanced Computing Center. Each compute node is configured with two Xeon E5-2680 processors for a total of 16 cores per node. Each node has 32GB of “host” memory. Nodes are interconnected with Mellanox FDR InfiniBand technology. We use 256 compute nodes (4,096 cores) for both collective and application...
Figure 7.9: Micro-benchmark Evaluations for Point-to-Point Optimization

Figure 7.10: Micro-benchmark Evaluations: Alltoallv

Figure 7.11: Micro-benchmark Evaluations: Allgatherv
evaluations. The results of the collective micro-benchmark are averaged with 1,000 iterations. The results of applications are averaged with multiple runs and are stable across different runs. All the codebase and applications are compiled with Intel Compiler 13.1.0.

7.5.2 Multi-endpoint Runtime Threads Latency

We evaluated the new Multi-Endpoint based runtime for multi-threading with the process-based runtime and the fine-grained, single-endpoint runtime through OSU Ping-Pong latency tests in Figure 7.8. For the process-based runtime (referred as PROCESS in the legends), we run the tests with pure MPI processes and the number of processes is equal to the number of cores on a single node. The Ping-Pong latency is measured between two nodes where processes on different nodes are doing back-to-back send/receive in pairs. For the SINGLE-ENDPOINT and MULTI-ENDPOINT, which refers to the fine-grained single endpoint runtime and the Multi-Endpoint runtime, there is one MPI process per node and the number of threads is equal to the number of cores. Each thread calls MPI Send/Recv functions in the same pattern as the PROCESS situation. The fine-grained MPI runtime (referred as SINGLE-ENDPOINT in the legends) provides 40us latency for a small message. By reducing the overhead of locks, threads can make MPI communication requests as efficient as processes in the Multi-Endpoint runtime.

7.5.3 Point-to-Point Micro-benchmark

We apply the point-to-point optimization on the non-blocking latency test. In the master-only mode, only the master thread makes MPI_Isend/MPI_Irecv function calls to an MPI rank on the other node. In the Multi-Endpoint mode, we put the MPI communication functions including MPI_Isend, MPI_Irecv, and MPI_Waitall functions inside the OpenMP
region according to the sample code shown in Section 7.3. We compare the performance across different number of active endpoints with the default single-endpoint runtime.

In Figure 7.9, according to the number of active endpoints, the latency varies depending on the message range. For message sizes that are less than 256 byte, the message division overhead can not be covered by the benefits, where “Endpoint = 1” gives the best performance; From 256 byte to 32K, “Endpoint = 2” is able to provide the best performance; “Endpoint = 4” performs the best for large message range. The Multi-Endpoint optimization can reduce the latency of the master-only model by up to 40%. The Endpoint Controller is configured according to the evaluation results, so it can choose the suitable number of endpoints according to the message size. In the application evaluations, we apply the Endpoint Controller for the best results.

7.5.4 Collective Micro-benchmark

We use the OMB microbenchmark for the evaluation of the transparent support to collective operations based on the new Multi-Endpoint runtime.

For process-based alltoallv operation, there are three patterns to complete the pair-wise algorithm. With a “blocking” pattern, each MPI rank finishes the send/receive requests to a certain destination before launching send/receive requests to other destinations. With a “flood” pattern, each MPI rank uses non-blocking send/receive operations to launch all of the communication requests for all of the destinations and then wait for them to complete. With a “window-based” pattern, the MPI rank launches the number of send/receive requests equal to a window size and wait for the posted requests to finish, before starting the next window of requests. The best performance is achieved when it switches between the three patterns regarding the scale and the number of MPI ranks on the node. We compare our
Multi-Endpoint optimized “Group” algorithm, as introduced in Section 7.4, with the best results from the three patterns (in this case, “flood” pattern is doing the best). Also, the Multi-Endpoint runtime automatically adjusts the number of active endpoints according to the message size and the number of total MPI ranks, according to a pre-defined tuning table.

As shown in Figure 7.10, we can observe that the new Multi-Endpoint runtime, referred as “mep” in the figure, can achieve 30% improvement for small messages. For large messages that are transferred through Rendezvous protocol, the new runtime can achieve up to 14% improvement.

Similarly, we also evaluate allgatherv micro-benchmark for the process-based single-ring algorithm, process-based double-ring algorithm, and multi-endpoint-based double-ring algorithm as introduced in Section 7.4. The evaluation starts from 1K message size because that for small messages Bruck algorithm can achieve better performance [31]. From 1K to 16K, the multi-endpoint, double-ring algorithm can achieve up to 30% improvement compared with process-based, double-ring algorithm. For large messages, the multi-endpoint runtime can keep a 15% to 22% improvement.

7.5.5 Application Evaluations

We evaluate the impact of the new Multi-Endpoint runtime with several application benchmarks written by outside researchers.

7.5.5.1 NAS Benchmark

NAS benchmark is a popular application benchmark set for High Performance Computing evaluations [23]. We include the hybrid MPI/OpenMP version of CG, LU, and MG in this evaluation. For the Multi-Endpoint runtime, we updated the benchmarks according
to the point-to-point optimization as introduced in Section 7.3. We carried out the tests for NAS benchmarks on 256 nodes with 4,096 cores. Each node is occupied by one MPI rank and each MPI process spawns 16 OpenMP threads. We compare the performance of the new Multi-Endpoint runtime with the default single-endpoint codebase in Figure 7.12. The Multi-Endpoint runtime with point-to-point optimization can achieve 6.3% improvement for MG, 11.7% improvement for CG, and 12.6% improvement for LU.

![Figure 7.12: NAS Benchmark on Stampede](image)

![Figure 7.13: Increased memory footprint per endpoint on Stampede](image)

7.5.5.2 P3DFFT Kernel Lib

The second application benchmark is the driver_sine program based on MPI/OpenMP hybrid version of the Parallel Three-Dimensional Fast Fourier Transforms (P3DFFT) library [95]. P3DFFT Kernel Lib from the San Diego Supercomputer Center (SDSC) is a portable, high performance, open source implementation based on the MPI programming model. It leverages the fast serial FFT implementations of either IBM’s ESSL or FFTW. P3DFFT uses a 2D, or pencil, decomposition and overcomes an important limitation to
<table>
<thead>
<tr>
<th>Num. of cores</th>
<th>Matrix Size</th>
<th>original</th>
<th>mep</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,048</td>
<td>2,048</td>
<td>3.62</td>
<td>3.34</td>
<td>7.7%</td>
</tr>
<tr>
<td>4,096</td>
<td>2,048</td>
<td>1.70</td>
<td>1.60</td>
<td>5.9%</td>
</tr>
<tr>
<td>4,096</td>
<td>4,096</td>
<td>13.80</td>
<td>11.93</td>
<td>13.5%</td>
</tr>
</tbody>
</table>

Table 7.1: P3DFFT Performance on Stampede (Time: sec)

scalability inherent in FFT libraries by increasing the degree of parallelism up to $N^2$, where $N$ is the linear size of the data grid. It has been used in various Direct Numerical Simulation (DNS) turbulence applications [46]. The test program initializes a 3D array of real numbers as a sine wave and calls forward real-to-complex FFT and then backward complex-to-real FFT. It is executed on 128 and 256 compute nodes, which includes a total of 2,048 and 4,096 cores. We run the test with one MPI rank per node and 16 OpenMP threads with each MPI process. The dimensions of the boxes are $2,048 \times 2,048 \times 2,048$ and $4,096 \times 4,096 \times 4,096$. The number of dimensions for processor decomposition is tow and the processor grid is $8 \times 16$ with 128 nodes and $16 \times 16$ with 256 nodes. We compare the Multi-Endpoint runtime performance with the original single-endpoint performance in Table 7.1. The P3DFFT Kernel Lib depends heavily on MPI_Alltoallv for the parallel FFT algorithm. We are able to observe a 30% improvement in communication time and 13.5% improvement in the total execution time.

### 7.6 Discussion

In this chapter, we either have OpenMP threads directly access runtime (Section 7.3) by simply adding OpenMP pragma outside MPI functions, or we wake up OpenMP threads inside the MPI runtime for collective operations in order to provide transparent support to MPI/OpenMP applications (Section 7.4). In both situations, OpenMP threads are the only
multi-threading model that executes MPI communication requests. There is another way to utilize the Multi-Endpoint runtime by spawning helper threads (different from the OpenMP threads) by the MPI runtime. The MPI functions are executed by master OpenMP threads while inside the runtime, the helper threads are launched for communication. However, in the helper-thread mode, the application has to inform runtime about the availability of cores to avoid oversubscribing, or the Operating System has to detect free cores and schedule the helper threads, which brings extra overheads. The hybrid group of the MPI Forum is working on an Endpoint proposal in order to expose the concept and control of endpoints to MPI applications for future hybrid programming models. Our design can be easily transplanted for new MPI interfaces related to Multi-Endpoint that would likely appear in future MPI specifications.

Another issue regarding the Multi-Endpoint runtime is the memory footprint. In order to remove locks inside the critical components, we added thread-specific Progress Engines and communication resources. This increases the memory footprint for communications. However, as discussed in Section 7.2.1, the “matching” mode keeps the increase of communication resource requirements as $O(N)$, where $N$ is the equal to the maximum number of endpoints. On Stampede cluster, we increased the memory usage for communication on a single node from 108 MB to 200 MB with 16 active endpoints, by measuring a hello-world program with MPI_Barrier with VmSize. Figure 7.13 shows the increased memory footprint per endpoint. The memory footprint per endpoint increases from 5 MB to 10 MB as the number of cores increases from 32 to 4,096 (from 2 nodes to 256 nodes). Comparing with the 32 GB memory on a single node, we assume that the increase of memory footprint due to endpoints is acceptable on the modern multi-cores systems.
7.7 Related Work

MPI/OpenMP hybrid programming model is widely regarded as suitable model for scaling parallel applications on emerging multi-/many-core computing architectures. However, there are several challenges associated with combining these two models (MPI and OpenMP) and fully leverage the performance benefits of a hybrid model. While previous work has shown performance improvements with a hybrid model [29, 71], there have been many reports of poor performance [36]. Scientific applications need to be carefully redesigned to fully leverage the performance benefits of the hybrid programming model [80].

Modeling and analyzing the performance of MPI/OpenMP applications has been an active area of research [20, 22, 32, 44, 53, 106]. Recently, authors have also explored the challenges of improving the performance of hybrid MPI/OpenMP applications on emerging accelerators [25, 96]. Li et al., have explored the potential benefits of mapping MPI ranks to threads to improve the performance of collective operations [75]. Recently, MPI libraries, such as MPICH, have introduced support for fine-grained locking mechanisms to improve the performance of such applications [24, 50, 52, 58, 59]. However, as authors note in [24], some of the critical operations within the MPI library still need to be serialized and the performance of thread-based communication operations may incur performance overheads when compared to process-based operations. Previously, we proposed a Multi-Endpoint based communication runtime for PGAS models, such as UPC [78]. In this chapter, we propose a lock free MPI communication runtime based on the Multi-Endpoint design. We demonstrate that our designs can offer thread-safety without relying on expensive locks and can significantly improve the communication performance of hybrid MPI/OpenMP applications.
7.8 Summary

In this chapter, we proposed a novel design of lock-free, Multi-Endpoint runtime for MPI/OpenMP hybrid applications. This design can efficiently utilize the compute and network resources and support dynamic communication optimization through Endpoint Controller to boost the performance for hybrid parallel applications with minimal modifications. Especially for MPI collective-based, parallel applications, our design enables those applications to directly and transparently increase their performance. Our evaluation demonstrates that our design can improve the performance of the popular scientific kernels based on the MPI/OpenMP paradigm. We demonstrate that our designs can improve the performance of the popular scientific kernels based on the MPI/OpenMP paradigm, such as CG, LU, MG, and P3DFFT by up to 14% with 4,096 cores.
Chapter 8: IMPROVING GPU TO GPU COMMUNICATION FOR
UPC RUNTIME OVER INFINIBAND

As mentioned in Chapter 3, accelerators and Co-Processors are getting more and more popular in HPC area, with new contention issues on communication resources. In the next two chapters, we extend our study with multi-endpoint design proposed in previous chapters on these heterogeneous architectures.

GPGPU accelerators are becoming more and more attractive in many scientific fields. The involvement of device memory has complicated the memory architecture of modern multi-core system. Without support from programming model and the middleware below it, accelerators increases the complexity of programming, as well as decreasing the communication efficiency over the network. If we take UPC+CUDA model for example, users have to manage the device memory which is outside the global address. They also need to use temporary host buffers and CUDA functions to explicitly transfer data between GPU to GPU.

In this chapter, based on the multi-threaded UPC runtime we proposed in Chapter 6, we introduce multiple extensions in order to 1) support remote GPU device memory access through standard UPC APIs; 2) provide advanced GPU to GPU communication based on the RDMA features of InfiniBand; and 3) reduce the latency of remote device memory access, when the remote UPC threads are busy with kernel functions or I/O operations.
We discuss the design challenges and solutions for the new runtime. Then we evaluate its performance through micro-benchmarks and a sample application.

The rest of this chapter is organized as follow: In Section 8.1, we provide further motivation on this work. In Section 8.2, we present the design details for the proposed new runtime to realize advanced GPU to GPU communication over InfiniBand. In Section 8.3, we evaluate the performance and benefits of the new runtime, through both micro-benchmark and sample application test. Related works are discussed in Section 8.4. Finally we conclude this chapter in Section 8.5.

### 8.1 Motivation

```c
cudaMalloc(&deviceBuffer);
shared_temp_send_buffer = upc_all Alloc(...);
shared_temp_recv_buffer = upc_all Alloc(...);
...
/*communication starts*/
if(iamsender) {
    cudaMemcpy(local_temp_send_buffer, deviceBuffer,...);
    /*make sure remote temporary buffer can be written in*/
    upc_barrier;
    upc memput(remote_temp_recv_buffer, local_temp_send_buffer, size);
    /*synchronize with receiver for end of data transfer*/
    upc_barrier;
} else if(iamreceiver) {
    /*make sure local temporary buffer are free from previous communication*/
    upc_barrier;
    /*make sure data already arrived*/
    upc_barrier;
    cudaMemcpy(deviceBuffer, local_temp_recv_buffer, ...);
}
```

Figure 8.1: UPC and CUDA Operations for Manual Remote Device Memory Access
Figure 8.1 illustrates an example to program on a heterogeneous cluster with GPGPUs using UPC and CUDA. Since current UPC software doesn’t support the global shared address space in GPUs, users need to call cudaMemcpy functions in order to allocate memory on GPU device and cudaMemcpy functions in order to move data between host memory and device memory. This increases the complexity of the memory management at user-level. From the sample code, we can find that point-to-point synchronizations are needed to guarantee that the host memory is available before upc_memput or upc_memget functions. It is also needed to inform the remote side to start cudaMemcpy. We use upc_barrier function for synchronization purpose in the pseudocode, due to lack of point-to-point synchronization functions in current UPC. It is possible to replace the upc_barrier functions with more complicated point-to-point synchronization implementations in the real codes. The synchronizations can increase the execution time at user-level. Moreover, since the explicit memory movement between CPU and GPU is necessary, both the destination and the source UPC threads have to be involved in the communication explicitly. If one of them is busy computing, it will lead to the waiting status on the other side.

In order to solve these problems that we have observed in current UPC and CUDA hybrid applications, we propose a new multi-threaded runtime which can support shared segments on both host memory and device memory. It supports GPU device memory access through the standard UPC APIs; furthermore, each UPC thread can get shared pointers from both device memory and host memory in a single run of an application. The new multi-threaded runtime provides more efficient communication, through multiple advanced designs based on the RDMA features of InfiniBand interconnects. By the
multi-threaded nature, the new runtime provides low latency remote device memory access through runtime-level helper thread, even when the remote UPC threads are busy with kernel calculations.

8.2 Design

In this section, we discuss the design details of realizing advanced GPU to GPU communication through the multi-threaded UPC runtime.

8.2.1 GPU Global Address Space for Multi-threaded Runtime

In UPC, each UPC thread is associated with partitioned global address space. The host memory is divided into shared segment and private segment. Variables in shared segment can be accessed by other threads directly. In previous design [111], though the global address space spans both host memory and device memory, a UPC thread has only one shared segment either on host memory or in device memory. However, current GPU applications are requiring calculations from GPU and CPU at the same time. Shared segment on either host memory or device memory will limit the flexibility of the applications. Thus, in this design, we keep two shared spaces on both device memory and host memory separately. Pre-allocated GPU device memory is divided into shared segment and private segment, as well as host memory. To allow fast distribution of the distributed data structure on device memory, the shared segment of device memory is maintained by the memory allocation data structures residing on host memory. Through extended APIs ‘upc_on_device’ and ‘upc_off_device’, the corresponding UPC threads will know where to allocate the buffer. Pseudocode can be found in Figure 8.2. After calling upc_on_device function, all the following UPC shared memory allocation functions will return a pointer-to-shared which points to a shared memory address allocated on device memory. By calling upc_off_device
function, UPC threads now get shared memory address on host memory again with UPC shared memory allocation functions. For multi-GPU situation, UPC threads are bounded to different devices in the initialization stage. The device shared segment and device private segment of a certain UPC thread then will be pre-allocated on its associated GPU device.

```c
... upc_on_device();
upc_all_alloc(nblocks, nbytes); // allocated on device shared segment
upc_alloc(nbytes); // allocated on device shared segment
...
upc_off_device();
upc_all_alloc(nblocks, nbytes); // allocated on host shared segment
...```

Figure 8.2: Pseudocode for Shared Memory Allocation with Extended APIs

For multi-threaded runtime, the pre-allocation of device memory is executed by Thread 0 on each node. Later, it is logically divided into portions and is linked with other threads. In multi-GPU situation, the affinity of threads and device memory portions will be considered according to the affinity of GPU device and physical processor associated with a given pthread.

### 8.2.2 Design for Remote Memory Operation

In this section, we discuss the implementation details about remote device memory access operations within the extended runtime, including upc_memput and upc_memget. When the GPU to GPU communication is hidden from application users, one of the most important issues is how to keep efficient data transfer over network. In the basic manually implemented communication code as shown in Figure 8.1, the user needs to explicitly
claim temporary buffer and add barriers for synchronization purpose. Since the claimed temporary buffer on host memory is inside shared segment, which is pre-pinned-down with InfiniBand NIC, the following upc_memput or upc_memget functions can be directly mapped to RDMA WRITE/READ operations.

One of the goals of the extended runtime is to free users from explicit management of communication buffers and barriers. GPU shared segment implementation helps releasing remote side at application level. However, we still want the pure communication performance being as efficient as the user-level two-sided mode, or even better. Thus, we utilize RDMA Fastpath design [76] to provide low-latency for small and medium data transfers.

In this design, both the local and remote side keep a bunch of paired and pre-pinned-down buffer chunks. Each chunk is associated with another chunk at the remote side, which means the rkey and address on remote side are known to the local thread. In order to avoid wasting of memory due to pair-wise design, the RDMA Fastpath buffer for GPU data transfer is only allocated when there is communication launched between two UPC threads. As shown in Figure 8.3(a), for upc_memput operation, the caller of the function will first check if there is available chunks inside RDMA Fastpath buffers. If the buffer is available, it will grab the next free chunk indicated by a head pointer and copy data from device memory to the data section inside this RDMA Fastpath buffer chunk. Information about the remote device memory address, which is translated from pointer-to-shared, will be wrapped into the message header before the real data payload. After the data movement is done, the caller will place a RDMA WRITE request on its endpoint for this Fastpath buffer, according to its associated rkey and remote address. The caller then can return from this upc_memput function since the original buffer on GPU is ready for re-use. In the whole process, the caller doesn’t need to be blocked by the status of the remote side. At
the receiver side, whenever the receiver goes to poll the network for incoming messages, it will get a chance to check the current head pointer of RDMA Fastpath buffer. Once a new message is detected in the head chunk, it will check the header of the whole message for the destination information. Then it will start cudaMemcpy to move the real data payload from RDMA Fastpath receive buffer to device memory.

Figure 8.3(b) shows the mechanism of RDMA Fastpath buffer based memget operation. For memget, the caller has to be blocked until the data is fully arrived inside the destination address. First of all, caller of the function will send an ACK message to the remote side, including the source address (which is on remote device memory and is translated from shared pointer to local pointer through shared segment address management functions) and size of the data. When the remote UPC thread detects the ACK message, it will get a chunk of free RDMA Fastpath buffers and do cudaMemcpy from the source address on device memory to the buffer. An RDMA WRITE request then is placed to the remote side endpoint. After the original caller detects incoming message inside its RDMA Fastpath buffer, cudaMemcpy is called to move data from the buffer to destination on local device memory.

The clearance of RDMA Fastpath buffer is done through piggy-backing data on regular packet exchange between the two UPC threads. When there is only one-way communication, the destination thread will send separate ACK messages back when the number of consumed buffers crosses the half watermark.

In the case where the message is too large to fit into RDMA Fastpath buffers or when the RDMA Fastpath buffers are drained up, another scheme is applied. The caller needs to allocate and pin-down temporary buffer inside runtime and send ACK message to remote side which includes the rkey of the temporary buffer, size and the address on remote device
memory, as shown in Figure 8.4. After the remote side detects ACK message, it will also allocate and pin-down temporary buffer for RDMA READ/WRITE operations, for the corresponding memput/memget functions accordingly. The remote side will send REPLY message back when it finishes operation on remote temporary buffer. Memput function callers can directly return and release the temporary buffer when it comes back to poll network channel and get REPLY message later. For memget function callers, it will block waiting until the REPLY from remote is received. Then it can do cudaMemcpy and release the temporary buffer before returning. In order to reduce the overhead from malloc and pin-down operations, we further apply registration cache design [82] and keep a list of free-delayed buffers. In addition, overlapping of communication and data movement between host memory and device memory [104] can be achieved for large messages too.

### 8.2.3 Helper Thread for Improved Asynchronous Access

As discussed in Section 8.1, the remote side processors have to be involved in the remote device memory access operation, without support from hardware. As in previous
Figure 8.4: UPC/CUDA Remote Device Memory Operation for Large Message

studies and discussion in Section 8.2.1, remote side involvement is shifted from application-level to the underlying runtime. However, when the remote UPC threads are busy with kernel function calls or even I/O operations, the remote device memory access has to be delayed. This is true even when the required destination or source address on device memory is free to be accessed. For message passing based programming model such as MPI, this is not a very serious problem since the nature of MPI operations is two-sided based and the applications adapting MPI are usually having balanced workload on every MPI rank. However, for PGAS languages such as UPC, direct access is the nature of this programming model which makes it a favorite of irregular applications. The requirement of remote side involvement brings limitations for such kind of applications. The straight-forward solution is helper thread. In optimized CUDA applications, user usually manages helper thread to achieve better overlap of computations and communications.

Based on multi-threaded UPC runtime, we realize true runtime helper thread to handle incoming device memory access automatically. When the automatic helper thread scheme is turned on as a runtime parameter, a dedicated helper thread will poll the endpoints of UPC threads who are busy with GPU kernel functions (Figure 8.5). This is possible even
with multi-GPU situation, since with multi-threaded runtime, the multi-endpoints associated with different UPC threads are inside the same process address space. When the helper thread detects a device memory access on an address which belongs to a busy UPC thread, it will do cudaMemcpy as representative for the original receiver of the request. For memput operations, the helper thread can reduce the total latency on destination side from $T_{cal} + T_{cpy}$ to $\max(T_{cal}, T_{cpy})$, where $T_{cal}$ indicates the time of kernel function and $T_{cpy}$ indicates time for data movement between device and host memory. For memget operations, not only the remote side, but also the latency at the caller side can be reduced by $T_{cal}$. However, in order to achieve data integrity and the correct order of the operations, the helper needs to lock its current working endpoint and also keeps a pending work list in case the original UPC threads return from kernel function when the helper thread is operating on the remote device memory access.

Figure 8.5: Dedicated Helper Thread Carry out Device Memory Access for Busy UPC Thread
8.3 Evaluations

8.3.1 Experimental Platforms

The evaluation is executed on a cluster with four nodes equipped with dual Intel Xeon Quad-core Westmere CPUs operating at 2.53 GHz and 12 GB of host memory. These nodes have Tesla C2050 GPUs with 3 GB DRAM. The InfiniBand HCAs used on this cluster are Mellanox QDR MT26428. Each node has Red Hat Linux 5.4, OFED 1.5.1, and CUDA Toolkit 4.0. Our work is based on the multi-endpoint supported UCR [78], which supports the latest Berkeley UPC via GASNet [28]. Performance of all micro-benchmarks are averaged over 1,000 iterations.

8.3.2 Micro-benchmark Evaluation

We first evaluate the micro-benchmark level performance of the new UPC runtime by upc_memput and upc_memget for GPU to GPU communication over network. The destination address of upc_memput and the source address of upc_memget functions are shared pointers returned by upc collective allocation operations after UPC threads call upc_on_device API. Thus the shared pointers are associated with shared segment on GPU device memory. The local source/destination address of upc_memput/upc_memget operations is a local pointer to local GPU device memory. The upc_memput function writes specified amounts of data bytes to the remote device memory according to local device memory. The upc_memget function fetches specified amount of data bytes from remote device memory to local device memory. We compare the new runtime performance with the user-level GPU to GPU communication micro-benchmark, which consists of explicit cudaMalloc, cudaMemcpy function, upc_barrier and normal upc_memput or upc_memget operation on host memory, as the same style used in pseudocode Figure 8.1.
Figure 8.6: Micro-benchmark Evaluations: memput latency

Figure 8.7: Micro-benchmark Evaluations: memget latency
We measure the average latency of the one-sided operations through the two micro-benchmarks with 1,000 iterations and the results can be found in Figures 8.6 and 8.7 for memput and memget, respectively. In the results, the user-level GPU to GPU communication is indicated as ‘user-level’ and the new runtime is indicated as ‘ucr-mth’. As shown in Figures 8.6(a) and 8.6(b), we can observe that, through RDMA Fastpath design, which gets rid of extra barriers before re-writing the temporary buffer in the user-level remote device memory access test, the new runtime can achieve 23% improvement by reducing the latency from 33us to 25us. This is because in the two-sided pattern of user-level remote device memory access, two explicit synchronizations are required. One is before the upc_memput, in order to make sure that the previous data in the remote temporary buffer is already drained-up; another one is after the upc_memput operation on temporary buffer is finished, to inform the remote side to start the cudaMemcpy. While for the new runtime supported GPU to GPU communication, the RDMA Fastpath design includes mechanism of safe RDMA WRITE/READ operations by piggy-backing information or explicit ack messages when the number of consumed chunks goes over half of the total number of the available chunks. The local cudaMemcpy and remote cudaMemcpy then can overlap, which result in a latency bounded by one cudaMemcpy overhead and network communication overhead.

On the other hand, for upc_memget operation, we carry out similar comparison as for upc_memput. However, as shown in Figures 8.7(a) and 8.7(b), the new runtime can not provide any improvement compared to the original user-level remote device memory access with small and medium messages. This is because for memget operation, the caller has to wait for the data to be fully placed in the destination buffer. Thus for both basic style or the new runtime, the latency is bounded by the time of two cudaMemcpy operations and
latency from synchronizations. However, we’d like to notice that, though there is no performance improvement for small messages, the new runtime still can provide a much better productivity comparing with the user-level based style, which is also true for upc_memput operations.

For large messages, through overlap of data movement and RDMA operations, we are able to see 5% to 20% improvement, as shown in Figures 8.6(c) and 8.7(c).

### 8.3.3 Helper Thread Evaluation

This part evaluates the benefits of dedicated helper thread as proposed in Section 8.2.3. In this benchmark, the local UPC thread calls upc_memget operation to read a piece of 8K byte data on the remote device memory. On the other side, remote UPC thread is blocked with a kernel function to mimic unbalanced workload. It needs to be noticed that the required data is not operated by the kernel function. Then we measure the latency of each upc_memget operation on the caller side.

We compare the performance of the user-based style, the new runtime without and with the dedicated helper thread enabled, which is represented by line ‘user-level’, ‘ucr-mth-helper’ and ‘ucr-mth-no-helper’ in Figure 8.8. From the figure, we can find out that, for the user-based benchmark and the new runtime without helper thread on, the latency will be the kernel calculation time plus a consistent overhead, which is the 8K message transfer time from remote device memory to local device memory. This is because the request on device memory can only be detected and launched by the UPC thread who is associated with the source device address. When the helper thread is turned on, it’s obvious that the latency is reduced to the pure communication time of the required data plus an overhead of locking
endpoint. The application writers can also launch helper thread explicitly to achieve the same performance. However, it will increase the coding complexity.

Figure 8.8: Helper Thread Evaluation with memput/memget

Figure 8.9: UPC/CUDA Matrix Multiplication

### 8.3.4 Sample Application Evaluation

In this section, we use a sample UPC/CUDA benchmark to evaluate the performance of the proposed new runtime. The evaluated benchmark realizes a parallel matrix multiplication algorithm, which calculates $C[N][N] = A[N][N] \times B[N][N]$. The matrix $B$ is divided into $n$ ($n$ equals to the number of GPUs) matrix $B_i[N][N/n]$ and $B_i$ is associated with UPC thread with thread ID $i$. The UPC threads will call GPU kernel function to do matrix multiplication $C_i[N][N/n] = A[N][N] \times B_i[N][N/4]$ and the result $C_i$ will be sent to UPC thread 0 for final summary. In the sample application for the new runtime, all the matrix elements are claimed through UPC allocation functions after upc_device_on and the final data transfer will be sent through upc_memput with both source and destination address on device memory. We also created a basic version of the sample application, which uses all CUDA
memory access and allocation functions explicitly. The basic version also needs to claim temporary buffer to hold the data from local device memory before sending it to remote host memory.

The sample application includes multiple iterations and the average time for each iteration is shown in Figure 8.9 for increasing matrix size. We break down the total time to computation time and communication time. Due to the high performance of accelerator and a relatively low speed for data transfer between device and host memory, we can observe that, the communication time consumes a big portion in the whole sample application. The new runtime can provide not only a good productivity for application users, but it can further improve the performance. For \( N = 50 \), the improvement is 34%. For \( N = 300 \), the improvement is 17%.

### 8.4 Related Work

There have been several studies related to UPC issues on GPU clusters from different levels. Chen et al. implemented a compiling system to extend UPC to take advantage of GPU clusters [40]. Through the proposed compiling system, affinity-aware loop tiling, GPU code generation and several memory optimizations targeting NVIDIA CUDA can be achieved. In [111], Zheng et al. discussed multiple implementation issues in supporting GPU for Berkeley UPC software stack, including unified APIs for data management and communication, end-to-end data movement, asynchronous task executions and so on. These extensions are based on runtime. We have a detailed discussion about this study in Section 8.1.
For other PGAS programming models, Lee extended XcalableMP for multi-node GPU clusters as XcalableMP-ACC in [74]. Chen et al. carried out a series of studies about dynamic load balancing issue for single- and multi-GPU in [41] and [42], respectively.

The similar programming complexity problem on GPGPUs cluster has been investigated in MPI programming model [64, 93, 104, 105]. Through extending the address space to GPUs device, popular MPI libraries such as MPICH2, OpenMPI, and MVAPICH2 can directly operate on the GPUs device memory. Compared with MPI programming model, PGAS programming models such as UPC targeted in this chapter can simplify the programming for the data communication and provide better support for one-sided communication. But on GPGPUs cluster, the benefits of PGAS are decreased due to the explicit data management and movement on GPUs device memory. In this chapter, we extend UPC to GPUs device memory space to resolve these problems.

8.5 Summary

In this chapter, we examined the problems with GPU to GPU communications in current UPC/CUDA applications without advanced runtime support. Based on the challenges related to productivity, programming model and performance, we proposed a multi-threaded UPC runtime to achieve both high-productivity and efficient communication for GPU to GPU communications over InfiniBand Clusters. The proposed UPC runtime supports GPU shared address allocation and distribution to allow direct access on remote device memory through the standard UPC APIs. After releasing the remote UPC threads from synchronizations of GPU to GPU communications at user level, we introduced the details of runtime support for communication between local and remote device memory addresses. We utilized advanced InfiniBand RDMA features to improve the performance
of data movement to/from device memory and data transfer over network. We also proposed dedicated helper thread in runtime to achieve efficient remote device memory access when the owner UPC threads of the remote address are busy. Our experimental results show that the proposed runtime can achieve 23% performance benefit for upc_memput operation, compared to the user-level GPU to GPU communication. For the micro-benchmark with unbalanced workload, the proposed helper threads can reduce the upc_memget operation to pure data transfer time. Through a sample parallel matrix multiplication benchmark, the new runtime successfully achieved 34% to 17% improvement for different matrix sizes.
Chapter 9: UPC ON MIC: EARLY EXPERIENCES WITH NATIVE AND SYMMETRIC MODES

Following Chapter 8, which focused on GPGPU architecture, we continue our research on another emerging architecture, the Co-Processors. Intel Many Integrated Core (MIC) architecture is steadily being adopted in clusters owing to its high compute throughput and power efficiency. The current generation MIC Co-Processor, Xeon Phi, provides a highly multi-threaded environment with support for multiple programming models. While regular programming models such as MPI/OpenMP have started utilizing systems with MIC Co-Processors, it is still not clear whether PGAS models can easily adopt and fully utilize such systems.

In this chapter, we discuss several ways of running UPC applications on the MIC architecture under Native/Symmetric programming mode. These methods include the choice of process-based or thread-based UPC runtime for native mode and different communication channels between MIC and host for symmetric mode. We propose a thread-based UPC runtime with an improved “leader-to-all” connection scheme over InfiniBand and SCIF [63] through multi-endpoint support. For the native mode, we evaluate point-to-point and collective micro-benchmarks, Global Array Random Access and NAS/UTS benchmarks. For the symmetric mode, we evaluate the communication performance between host and MIC.
within a single node. Finally, we provide several insights to optimize running UPC on hybrid systems with MIC. Through our evaluations, we explore the effects of scaling UPC threads on the MIC and also highlight the bottlenecks (up to 10X degradation) involved in UPC communication routines arising from the per-core processing and memory limitations on the MIC.

The rest of this chapter is organized as follow: We further discuss alternatives for running UPC on MIC in Section 9.2. Section 9.3 gives our experimental results. In Section 9.4, some important insights are discussed. Section 9.5 lists the related work. Finally we conclude this chapter in Section 9.6.

9.1 Motivation

On the Intel MIC, applications are usually run in one of the following modes:

1. **Native**: MIC can be used in a *native* many-core mode where the application runs only on the Co-Processor.

2. **Offload**: MIC can be also used in an *offload* accelerator mode where the application runs only on the host and offloads compute-intensive parts of code to the Co-Processor.

3. **Symmetric**: A *symmetric* mode is also offered on MIC where the application can be launched on both the Co-Processor and the host. This mode provides maximum control in the way applications are launched and allows application developers to take full advantage of the resources offered by the host and Co-Processor.

Several previous works [49, 91, 92] have explored running MPI applications on MIC. With both the *native* mode and the *symmetric* mode, the limited memory on the MIC places
constraints on the number of MPI processes that can run on it and hence can be prohibitive towards fully utilizing the hardware capabilities available on it. One way to overcome this limitation is to resort to hybrid programming models of MPI and OpenMP to minimize the process count and memory footprint, and allow for light-weight threads to fully subscribe all the cores on the MIC. However, this approach involves more programming efforts to modify MPI applications with OpenMP directives. Hence the availability of a highly multi-threaded environment and the need for high programming productivity makes PGAS implementations such as UPC a more natural fit for the MIC.

The offload mode allows programmers to use compiler directives to offload regions of computation that may benefit from acceleration on the Co-Processor. However, in this mode, the Co-Processor Offload Engine (COE) manages all memory operations with Co-Processor memory, and hence, any data residing on the Co-Processor must be first copied on to the host through COE if there is an intent to send that data out of the node. This increases the memory copy overheads on applications. Furthermore, programmers are not exempted from changing the source code of applications for marking the regions of computation to be offloaded. Also, for computations characterized by irregular communication patterns, it is hard to mark the regions of computation to be offloaded due to the irregular computation feature. Therefore, in this chapter we mainly explore and discuss the situations of evaluating the performance characteristics of UPC runtime and running UPC applications under native and symmetric modes. Even for these two modes, we still need to address and investigate additional problems, as outlined below:
1. UPC threads can be mapped to either OS process or thread. The process-based runtime and thread-based runtime rely on different schemes for intra-node communication. Taking the new features of MIC into consideration, what are the performance differences between these two runtime schemes?

2. As shown in Figure 9.1(a), the limited memory on the MIC leaves dozens of UPC threads sharing a small global memory region in the \textit{native} mode. What is the performance impact on UPC applications of such kind of many-core architecture with a small memory space?

3. As shown in Figure 9.1(b), in the \textit{symmetric} mode, the host memory space and Co-Processor memory space will be integrated as a global memory region for UPC threads running on both cost and Co-Processor. What are the performance characteristics of the communication between host and MIC?

We all of above issues and shares our early experiences of UPC on the MIC under \textit{native} and \textit{symmetric} modes.

\section{Alternatives for running UPC on MIC}

\subsection{UPC Runtime on MIC: Process-based or Thread-based?}

UPC \textit{thread} refers to an instance of execution for UPC applications. Depending on different implementations of UPC runtime, each UPC thread can be mapped to either an OS process or an OS thread. While OS threads share the entire address space, processes can only share certain memory regions. Thus, in existing process based runtime, the intra-node memory access between two UPC threads has to be realized through one of the two shared memory based schemes. The first scheme is copy-via-shared-memory. In this scheme,
shared memory performs as intermediate buffer. The owner of the source copies the data from source buffer to a temporary buffer that has been mapped to shared memory. Then the owner of the destination copies the data from the temporary buffer to the destination buffer. There are three disadvantages of the copy-via-shared-memory scheme: overhead from extra copy, extra memory footprint for intermediate buffer, and extra synchronization between source and destination processes. Many-core systems such as MIC have larger number of cores and limited memory compared to normal multi-core systems. The copy-via-shared-memory scheme thus suffers higher overhead on MIC. Furthermore, the synchronization between source and destination is also against the one-sided native feature of UPC programming language.

The second scheme utilized by process based runtime is shared-memory-mapping. The Berkeley UPC runtime offers a feature called PSHM (Inter-Process SHared Memory) [27].
PSHM provides support for UPC shared arrays using Unix System V (SysV) shared memory. The whole local part of the shared memory region, which belongs to one process, needs to be mapped into the space address of all the other processes. All other processes then can direct read/write UPC shared arrays through shared memory mapping. This removes the extra copy overhead and synchronization problems in the first scheme. However, the mapping of shared region from every process to all the other processes generates a huge number of memory footprint in the kernel space. The number of required entries in page table is $O(N^2)$, where $N$ equals to the number of processes. In a many-core systems like MIC, with possibly more than 60 processes, a large amount of memory is consumed for the page table. In short, the problems with the two shared memory based schemes on multi-core systems are worse off on many-core systems like MIC.

On the other hand, thread-based runtime can achieve much lower latency to directly access shared array with no extra memory requirement. This is because all the UPC threads on the same node are mapped to OS threads spawned by a single OS process. As a result, the global shared array region within a single node belong to the same address space. UPC direct memory access within a single node between different UPC threads then can be done by system memory copy functions directly from the source buffer to destination buffer.

Regarding to the drawbacks of thread based runtime, previous research [54] has reported that it suffers bad network performance, due to sharing of network connections when hardware allows one connection per process. Our previous work [78] proves that through multi-endpoint support, thread-based runtime can achieve same performance as process based runtime.

Based on the comparison between process based and thread-based runtime implementations for intra-node communication, in the many-core system with increasing intra-node
communication and limited memory space, thread-based UPC runtime is the direction from both performance and scalability point of view. We thus choose to utilize thread based runtime for the evaluations in the following sections of the chapter.

9.2.2 Remote Memory Access between MIC and HOST

Communications between MIC and Host can go through either SCIF-based [63], which based on PCIe, or IB-based channels. Both of these channels provide remote memory access (RMA) semantics. It has been previously shown that, for this configuration, IB is suited in the small message range and SCIF is suited in the large message range [92]. For IB-based communication in the UPC process-based runtime, every process on the MIC needs to establish a connection with every process on the Host and vice versa. This requires order of $N_{MIC} \times N_{HOST}$ number of connections, where $N_{MIC}$ and $N_{HOST}$ refer to the number of UPC threads on MIC and Host, respectively. With 60+ cores on MIC, the number of connections will increase significantly for a large value $N_{MIC}$. Hence the process-based runtime can place increased pressure on the limited memory residing on the Co-Processor.

In order to reduce the memory footprint for MIC-to-HOST communication without compromising performance, we propose a “leader-to-all” connection mode for multi-endpoint support [78] in the multi-threaded runtime.

Figure 9.2 shows how the “leader-to-all” connection mode works. First of all, the MIC leader thread $MIC_{leader}$ registers a shared memory region local to the whole MIC Co-Processor. After the registration, all other threads on MIC can access this shared memory region just pinned-down by $MIC_{leader}$. The Host leader thread $HOST_{leader}$ does the same registration. After $MIC_{leader}$ and $HOST_{leader}$ finish registration, those two leaders exchange their rkey of the pinned-down memory. Those two rkeys are distributed to all the threads.
on MIC and Host by the $MIC_{leader}$ and $HOST_{leader}$, respectively. Then all the threads on MIC (Host) establish connections with the remote leader thread on Host (MIC). When a thread $HOST_i$ needs to access a shared array address belonging to any thread $MIC_j$ on the MIC, thread $HOST_i$ uses the connection between itself and the thread $MIC_{leader}$ with the distributed rkey. Through this “leader-to-all” connection mode, the number of connections between MIC and Host is reduced to $N_{MIC} + N_{HOST}$.

For large message communication, where SCIF-channel performance is more optimal, we reuse the “leader-to-all” connection design through IB over SCIF mechanism [88].

### 9.3 Evaluations

In this section, we evaluate the proposed multi-threaded UPC runtime with micro-benchmarks and applications on MIC system with native and symmetric mode.

#### 9.3.1 Experimental Platforms

Our experimental environment is a dual socket node containing Intel Sandy Bridge (E5-2680) dual octa-core processors, running at 2.70GHz, a SE10P (B0-KNC) Co-Processor
and a Mellanox IB FDR MT4099 HCA. The host processors are running CentOS release 6.3 (Final), with kernel version 2.6.32-279.el6.x86 64. The KNC runs MPSS 4346-16 (Gold). The compute node is configured with 32GB of “host” memory with an additional 8GB of memory on the Xeon Phi Co-Processor card. The Intel compiler composer xe 2013.0.079 has been used. All the micro-benchmark results are averaged over 1,000 iterations.

The UPC implementation used in the evaluation section is based on Berkely UPC v2.16.0 and the multi-endpoint UCR runtime proposed in [78]. As discussed in Section 9.2.1, we have all the UPC threads mapped to OS threads and “leader-to-all” connection mode is utilized to enable communication between host and MIC in symmetric mode.

9.3.2 Micro-benchmark evaluations

In this section, we present the performance evaluation results of point-to-point and collective micro-benchmarks.

9.3.2.1 Native Mode Point-to-Point Evaluations

We first evaluate the native mode by running benchmarks purely on MIC and host respectively. “Intra-Host” refers to the results when running with native mode on host, where two UPC threads are all launched on host. “Intra-MIC” refers to the results with native mode on MIC.

In Figure 9.3, we compare UPC memput performance running with 2 threads on MIC or host separately. The latency of a 256 byte message size memput operation running on MIC and host are 0.2 and 0.01 µs, respectively. For large messages, a 1 MB message size
latency on MIC and host are 440 and 60 $\mu$s, respectively. We observe that there is a huge gap in the performance of the memput operation on the two platforms.

On the MIC, there is up to 10X memory copy overhead for a single communication pair in comparison with that on the host. This difference in performance can be attributed to the difference in performance of the `memcpy` operation as well as the frequency at which the cores run on these two platforms.

In Figure 9.4, we run the same benchmark with 16 threads on the host and with either 16 or 60 threads scattered on the MIC. For the MIC case, we have performed experiments with all 240 threads being used but as the performance was significantly worse, we have avoided including corresponding results. By using 16 threads on host and 60 threads on MIC, all of the cores on the CPU and Co-Processors are fully occupied by UPC threads. For small messages, the memput latency for 256 byte message size with 8 pairs of threads running on host and MIC are 0.01 and 0.2 $\mu$s, respectively. The memput latency for 256 byte message size with 30 pairs of threads running on MIC is 0.2 $\mu$s. We are able to observe that with increasing concurrent small message communication (8 pairs to 30 pairs), MIC is able to keep the same latency up to 256 KB message size. This trend can be explained by the high bandwidth ring available on the MIC [5]. For large message size, the memput latency for 1 MB message size with 8 pairs of threads running on host or MIC are 100 and 480$\mu$s, respectively. The memput latency of 1 MB message size with 30 pairs of threads running on MIC is 850$\mu$s. For large message size, the memput latency for 1 MB message size with 8 pairs of threads running on host or MIC are 100 and 480$\mu$s, respectively. The memput latency of 1 MB message size with 30 pairs of threads running on MIC is 850$\mu$s. We can observe that more concurrent large message transfers have more memory copy overhead, due to the limitation of the interconnects inside the MIC node.
Figure 9.3: Intra-MIC Evaluations: single pair memput latency

(a) small message

(b) large message

Figure 9.4: Intra-MIC Evaluations: multi-pair memput latency

(a) small message

(b) large message

Figure 9.5: Intra-MIC Evaluations: collective latency

(a) Bcast

(b) Gather

(c) Exchange
9.3.2.2 Native Mode Collective Evaluations

In this subsection, we evaluate the performance of UPC collectives running on host or MIC with varying number of threads. We chose Bcast, Gather, and Exchange as representatives for common patterns across HPC applications. In Figure 9.5(a), we present Bcast results with 16 threads on host, 16, 32, and 60 threads on MIC, respectively. We could see that there is performance difference between running 16 threads on host and MIC. When we increase the number of threads for collective operations, the performance of collective operations are affected by increased memory contention. In Figure 9.5(b) and 9.5(c), we run the same set of experiments for Gather and Exchange operation. We could see that Gather and Exchange operations have the same performance trends as Bcast operation.

9.3.2.3 Symmetric Mode Point-to-Point Evaluations

In this subsection, we evaluate the performance of memput operations across host and MIC in symmetric mode. We measure the performance from both host side and MIC side and show the results. In Figure 9.6, UPC threads launched on host CPUs initialize the upc_memput function calls while UPC threads on MIC are idle. We increase the number of threads involved in communication from 1 to 16. On other hand, UPC threads on MIC initialize the upc_memput functions in Figure 9.7, from 1 thread to 60 threads. When the number of threads on MIC and host increases, the memput latency increases as a result of memory contention. While single thread memput costs 0.38 \( \mu s \) and 16 concurrent memput costs 8.96 \( \mu s \) from host to MIC, as shown in Figure 9.6(a), MIC to host transfers perform significantly worse than host to MIC transfers as this involves the IB HCA reading from the MIC and this is known to suffer PCIe read bottleneck. In Figure 9.7(a), we observe that from MIC to host, memput costs 5.12 \( \mu s \) for single thread, 23.29 \( \mu s \) for 16 threads, and
56.88 $\mu s$ for 60 threads. This indicates some of the bottlenecks involved in transferring data over the PCIe as traffic increases.

![Graphs showing latency for small and large messages](image1)

Figure 9.6: Host-to-MIC Evaluations: multi-pair memput latency

![Graphs showing latency for small and large messages](image2)

Figure 9.7: MIC-to-HOST Evaluations: multi-pair memput latency

### 9.3.3 NAS Benchmark

In this section, we compare NAS benchmark performance with *native* mode on host or MIC. In Figure 9.8(a), we first test the NAS Class A performance with one pthread, which
means only one CPU on host or one Co-Processor on MIC is used for the execution of the benchmark. This figure shows the difference of the computation power between a CPU and a Co-Processor. For CG, EP, FT, IS, and MG, the execution times of a single Co-Processor on MIC is 42x, 5x, 11x, 7x, and 12x of a single CPU processor on host, respectively. Then, in Figure 9.8(b), we present the results of NAS Class B performance with full utilization of the host or MIC. In both host and MIC native mode, the CPUs and Co-Processors are fully occupied with 16 or 60 UPC threads respectively. We are able to observe that 60 MIC Co-Processors delivers 80%, 67%, and 54% performance as 16-CPU host for MG, EP, and FT. However, for communication-intensive benchmarks CG and IS, the MIC spends 7x and 3x execution time, due to the bottleneck of slower intra-MIC communication, as we observed in Section 9.3.2.1. Comparing with Figure 9.8(a), we find that the full utilization of 60 Co-Processors can improve the calculations time, while the new intra-MIC communication bottleneck can reduce this benefit.

Figure 9.8: NAS benchmark performance with native mode
9.3.4 Random Access

The Random Access benchmark is motivated by a growing gap in performance between processor operations and random memory accesses. This benchmark measures the peak capacity of the memory subsystem while performing random updates to the system memory. We carry on Random Access benchmark for native mode on host and MIC with fully utilization. With table size equal to 2,097,152, the host native mode got 93 $\mu$s while MIC native mode is 246 $\mu$s (Figure 9.9).

9.3.5 UTS Benchmark

The Unbalanced Tree Search (UTS) benchmark is a parallel benchmarking code that reports the performance achieved when performing an exhaustive search on an unbalanced tree [18]. In the benchmark upc_lock are frequently utilized for load balancing. Fetching request is implemented by upc_memget operation. We evaluate the native mode on Host and MIC and show the results in Figure 9.9. Due to the frequent small message exchange
introduced by upc_lock and upc_memget, we can observe 4X execution time on MIC than on Host.

9.4 Discussion

**Asymmetric performance in symmetric mode:** In Section 9.3.2.3, we observed that the Host-to-MIC and MIC-to-Host performance is not symmetric in *symmetric* mode. The HCA and PCIe bottleneck along with MIC Co-Processor results in much higher memory access overhead from MIC to Host memory. We further evaluated RandomAccess benchmark with *symmetric* mode with totally 76 UPC threads where 16 UPC threads on Host and 60 UPC threads on MIC. The execution time dramatically increases from less than 1 second in *native* mode (both Host and MIC) to 52 seconds. Similar performance degradation is observed with NAS-FT benchmark in *symmetric* mode. Four threads with *native* mode on Host takes 22 seconds for FT Class B test. However, when we launch the job in *symmetric* mode with four UPC threads and four UPC threads on MIC, the total execution time for FT benchmark is 116 seconds. Deploying delegate to handle communications from MIC to Host memory in a contention-aware manner could help alleviate the high asymmetric memory access overhead.

**Unbalanced memory and computation power:** As shown in Figure 9.1, MIC system has much less local memory (8G) than Host node (32G). In *symmetric* mode, when both of the Host memory and MIC memory are included in global shared array, the size of shared array region on Host must match the size of shared array region on MIC. The unbalance not only comes from memory region, but also the computation power of MIC. One of the benefits of *symmetric* and *native* modes is that UPC applications can run on MIC with no change to source code. This means Co-Processors are assigned with the same workload as CPUs.
However, considering the unbalanced physical memory size and unbalanced computation power, optimizations for UPC applications to assign corresponding workload alone with shared memory region on Co-Processors and CPUs would be an important consideration in this direction.

**Hyper Threading:** MIC has quad-HyperThreading for each Co-Processor. However, when we evaluated the performance of Hyper Threading by binding more than one thread on each Co-Processor, the performance drops dramatically. Future research work about utilizing Hyper Threading on MIC is remaining open.

### 9.5 Related Work

There have been many research to utilize MIC to accelerate parallel applications. Being a new terrain, optimal programming model for writing parallel applications on MIC is yet to be explored. In [91], Sreeram et al. presented their experience with MVAPICH2 MPI library on MIC architecture. They have tuned and modified the MVAPICH2 library to provide better performance for basic point-to-point and collective operations running on MIC architecture. Article [92] proposes designs that are based on standard communication channels and Intel’s low-level communication API (SCIF) to optimize MPI communications on clusters with Xeon Phi Co-Processors. They are able to show performance improvements of running applications on MIC. Besides existing works in MPI, there are also some works using UPC(Unified Parallel C) on MIC. Article [97] describes SGI compiler support for running UPC application on MIC.

Global Address Space Programming Interface (GASPI) [2] is a PGAS API. GASPI allows software developer to map memory of Xeon Phi to RDMA PGAS memory segments. Those segments could be directly read/write from/to between processes — inside node
and across nodes. Those RDMA operations could significantly minimize synchronization overheads. In this chapter, we explore the performance benefits of using UPC on MIC. We provide a detailed evaluation of UPC-level benchmarks and applications.

9.6 Summary

In this chapter, we discussed the intra-node memory access problems and Host-to-MIC connection issues for running UPC applications on MIC system under native and symmetric programming mode. We chose the multi-threaded UPC runtime with the new proposed “leader-to-all” connection mode according to the discussion. We evaluated point-to-point, collectives, Random Access test and NAS/UTS benchmarks for Native mode. We also examined the point-to-point communication performance between Host and MIC in symmetric mode. According to the evaluation results, we found out several significant problems for UPC running on many-core system like MIC and provided insights to possible future optimizations in order to take further utilization of MIC for UPC applications.
Chapter 10: SOFTWARE DISTRIBUTION

The major portion of the studies described in this dissertation has been incorporated into several software releases.

MVAPICH2 is an open-source MPI-2 implementation (conforming to MPI 2.2 standard and initial support for MPI-3) over InfiniBand, 10GigE/iWARP and RoCE. It delivers best performance, scalability and fault tolerance for high-end computing systems and high performance interconnects. This software stack has been utilized by more than 2,000 organizations world-wide in 70 countries. As of June 2013, MVAPICH2 software is powering several supercomputers in the TOP 500 list, include: 6th, 462,462-core (Stampede) at TACC, 19th, 125,980-core (Pleiades) at NASA, 21st, 73,278-core (Tsubame 2.0) at Tokyo Institute of Technology. Recently, MVAPICH2-X software package is released and it provides support for hybrid MPI+PGAS (UPC and OpenSHMEM) programming models with unified communication runtime for emerging exascale systems.

The details on software distribution are as follow:

- MVAPICH2 1.8 (April 2012) and after: The shared tail cyclic buffer design proposed in Section 4.2.1 is included in this release to improve MPI intra-node shared memory communication performance.
• MVAPICH2 2.0(TBD): The state-driven polling scheme proposed in Section 4.2.2 is going to be released with MVAPICH2 2.0 version in order to reduce polling overhead with increasing number of cores in the multi-core system.

The research related to UPC multi-endpoint design which are conducted in Chapter 6, Chapter 8, and Chapter 9 are going to be implemented in future MVAPICH2-X releases.
Chapter 11: CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

This chapter includes a summary of the research contributions of this dissertation and concludes with future research directions.

11.1 Summary of Research Contributions

The dissertation aims to design an efficient MPI and UPC runtime for multi-core clusters with InfiniBand, accelerators and Co-Processors, in order to address several critical shared memory and network contention issues. The successful research conducted in this dissertation contributes to efficient intra-node and inter-node communication for various programming models, including MPI, UPC, MPI/OpenMP, UPC/CUDA and so on.

We summarize our research contributions in detail in the rest of this section.

11.1.1 Improve MPI Inter-Process Communication over Shared Memory

In Chapter 4, we examined the current shared memory channel design in two popular MPI stacks: MVAPICH2 and MPICH2-Nemesis. Based on the three scalability issues within current designs, we proposed three new schemes for MPI shared memory communication: “Shared Tail Cyclic Buffer” is aimed at reducing extra access to shared control structures; “State-Driven Polling” scheme is used to reduce polling overhead with more and
more local destinations for pair-wise buffers; “On-Demand Global Shared Buffer Pool” is proposed to reduce contentions in accessing shared send buffer and shared receive queues and provide an efficient memory usage for large message transfer. We are able to see significant improvement for point-to-point intra-node communication performance in both micro-benchmark and application level evaluations.

11.1.2 Congestion Avoidance on Multi-core High Performance Computing Systems

In Chapter 5, we showed that contemporary networks or runtime layers are not very well equipped to deal with a large number of operations in flight and suffer from congestion. We distinguished two types of congestion: Rate Congestion happens when tasks inject too many concurrent messages, while Concurrency Congestion happens when too many cores are active at the same time. We proposed a runtime design using proactive congestion avoidance techniques: a thin software layer is interposed between the application and the runtime to limit the number of concurrent operations. This approach allows the communication load to increase to the point where native congestion control mechanisms might have been triggered, without actually triggering them. We implemented a congestion avoiding runtime for one-sided communication on top of two UPC runtimes for two networks: InfiniBand and Cray Gemini. We discussed heuristics to limit the number of messages in flight and present implementations using either task inline or server based mechanisms. The runtime can provide performance and performance portability for all-to-all collectives, fine grained application benchmarks, as well as implementations of the NAS Parallel Benchmarks.
11.1.3 Reducing Network Contention of Multi-threaded UPC Runtime through Multi-endpoint

In Chapter 6, we explored multiple design alternatives and the overall design space for UPC runtime implementation on multi-core architectures. Based on the evaluation and comparison between OS-process mapping and OS-thread mapping schemes, we investigated and presented a new multi-threaded runtime with multiple network endpoint design in Unified Communication Runtime (UCR). The new multi-threaded UCR-endpoint design is not only an improved version of UCR to enable multi-threading, but it also provides significantly better performance than existing GASNet multi-threaded runtime. We also presented novel benchmarks that attempt to capture communication and computation patterns of irregular applications. The results from these novel benchmarks demonstrate that our design is able to provide significant improvement to these irregular benchmarks with the aid of ‘communication helper threads’ and ‘work stealing’ concepts. These schemes provide runtime-level load balancing, which can only be utilized by thread-based runtime on multi-core architectures.

11.1.4 High Performance Lock-free Multi-endpoint Runtime for MPI OpenMP Hybrid Applications

In Chapter 7, we proposed a novel design of lock-free, Multi-Endpoint runtime for MPI/OpenMP hybrid applications. This design can efficiently utilize the compute and network resources and support dynamic communication optimization through Endpoint Controller to boost the performance for hybrid parallel applications with minimal modifications. Especially for MPI collective-based, parallel applications, our design enables those applications to directly and transparently increase their performance. Our evaluation
demonstrates that our design can improve the performance of the popular scientific kernels based on the MPI/OpenMP paradigm.

11.1.5 Improving GPU to GPU Communication for UPC Runtime over InfiniBand

In Chapter 8, we examined the problems with GPU to GPU communications in current UPC/CUDA applications without advanced runtime support. Based on the challenges related to productivity, programming model and performance, we proposed a multi-threaded UPC runtime to achieve both high-productivity and efficient communication for GPU to GPU communications over InfiniBand Clusters. The proposed UPC runtime supports GPU shared address allocation and distribution to allow direct access on remote device memory through the standard UPC APIs. After releasing the remote UPC threads from synchronizations of GPU to GPU communications at user level, we introduced the details of runtime support for communication between local and remote device memory addresses. We utilized advanced InfiniBand RDMA features to improve the performance of data movement to/from device memory and data transfer over network. We also proposed dedicated helper thread in runtime to achieve efficient remote device memory access when the owner UPC threads of the remote address are busy.

11.1.6 UPC on MIC: Early Experiences with Native and Symmetric Modes

In Chapter 9, we discussed the intra-node memory access problems and Host-to-MIC connection issues for running UPC applications on MIC system under native and symmetric programming mode. We chose the multi-threaded UPC runtime with the new proposed “leader-to-all” connection mode according to the discussion. We evaluated point-to-point,
collectives, Random Access test and NAS/UTS benchmarks for Native mode. We also examined the point-to-point communication performance between Host and MIC in symmetric mode. According to the evaluation results, we found out several significant problems for UPC running on many-core system like MIC and provided insights to possible future optimizations in order to take further utilization of MIC for UPC applications.

11.2 Future Research Directions

HPC area is evolving fast and the runtime for HPC programming models is driven by the new trend of emerging architectures. In this dissertation many designs are proposed to address performance and scalability issues from new multi-core/many-core clusters with high performance interconnects, for several different programming models including MPI, MPI/OpenMP, UPC, UPC+CUDA. There are still several research areas that are left to be explored.

**Unified Congestion Control Design:** In Chapter 6, we proposed dedicated helper thread and work stealing schemes to achieve load balancing in multi-endpoint UPC runtime. In Chapter 5, we presented proxy (server) congestion avoidance scheme in order to solve the network congestion with multi-core system. In the future, multi-core/many-core systems are expected to have even larger number of active cores. It is necessary to investigate the network contention problem in the environment of future multi-core/many-core system and design a unified congestion control design, such as runtime delegates, to improve the network communication performance regardless of the high-level application and programming models.

**Advanced Heterogeneous Architecture:** Co-Processors and accelerators architectures are steadily being adopted in HPC clusters. Systems equipped with multiple GPGPU
and MIC Co-Processors start to be deployed for even higher parallelisms and power efficiency. Tianhe-2 [51], which ranked No. 1 in the latest Top500 list (June 2013), is equipped with two MIC Co-Processors per node. The advanced heterogeneous architecture along with increasing number of CPUs introduces even more complicated hierarchy inside a single compute node. Future research efforts can be taken to investigate the contention problems and application optimization in the environment of advanced heterogeneous architectures.
Bibliography


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


[82] F. Mietke, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm. Reducing the Impact of Memory Registration in InfiniBandTM.


