SCALABLE JOB STARTUP AND INTER-NODE COMMUNICATION IN MULTI-CORE INFINIBAND CLUSTERS

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Jaidev K. Sridhar, B.E.

* * * * *

The Ohio State University

2009

Master's Examination Committee:

Prof. Dhabaleswar K. Panda, Adviser

Prof. P. Sadayappan

Approved by

Adviser

Graduate Program in Computer Science and Engineering

ABSTRACT

Moores Law – frequency scaling and exploitation of Instruction Level Parallelism due to increasing transistor density no longer leads performance gains in modern systems due to limitations in power dissipation. This has led to an increased focus on deriving performance gains by taking advantage of Data Level Parallelism through parallel computing. Clusters – groups of commodity compute nodes connected via a modern interconnect have emerged as the top supercomputers in the World and the Message Passing Interface (MPI) has emerged as the *de facto* standard in parallel processing models on large clusters. Scientific and financial applications have ever increasing demands for compute cycles and the emergence of multi-core processors has driven an enormous growth in the cluster sizes in recent years. InfiniBand has emerged as a popular low-latency, high-bandwidth interconnect of choice in these large clusters.

With cluster sizes continuing to scale, the scalability of MPI libraries and associated system support and resources such as the job launcher have been at the center of attention in the High Performance Computing (HPC) community.

In this work we examine the current job launching mechanisms that have scalability problems on large scale clusters due to resource constraints as well as performance bottlenecks. We propose a Scalable and Extensible Launching Architecture for Clusters (ScELA) that scales to modern clusters such as the 64K processor TACC Ranger.

We also examine the scalability constraints with point-to-point InfiniBand channels in MPI libraries. We use the eXtended Reliable Connection (XRC) transport available in recent InfiniBand adapters to design a scalable MPI communication channel with a smaller memory footprint. The designs proposed in this work are available in both MVAPICH and MVAPICH2 MPI libraries over InfiniBand, which are used by more than nine hundred organizations around the World.

This is dedicated to my parents

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. Dhabaleswar K Panda for his guidance throughout the duration of my M.S. study. The guidance and advice he provided during the course of developing this thesis and during the course of my M.S. program proved invaluable. I am thankful to Prof. P. Sadayappan for agreeing to serve on my Masters examination committee.

I am also thankful for the National Science Forum for the financial support they provided for my graduate studies and research.

I'm especially thankful to the senior members of NOWLAB, Matthew Koop, Gopal Santharaman, Dr. Wei Huang, Dr. Saundeep Narravula for their mentorship and guidance. I would also like to thank all my colleagues at NOWLAB – Lei Chai, Tejus Gangadharappa, Karthik Gopalakrishnan, Krishna Kandalla, Ping Lai, Miao Luo, Xiangyong Ouyang, Greg Marsh, Hari Subramoni for their support and making my stay thoroughly enjoyable. I would also like to thank Mr. Jonathan Perkins for being an excellant collaborator and technical contributor to my work in addition to his timely support with experimental equipment.

A lot of people made my years at the Ohio State University memorable, I would like to thank all of them.

Finally, I would like to thank my parents, M K Sridhar and P Kanakalatha, and my brother Jayanth Sridhar their constant and unwaivering love and support without which I would not have made it this far.

VITA

June 13, 1981	Born - Bangalore, India
2003	B.E. Computer Science & Engineering Visveswaraiah Technological University, Belgaum, India
2003-2003	Software Engineer, Hewlett-Packard ISO.
2003-2005	Software Development Engineer, NetScaler Pvt. Ltd.
2005-2007	Software Development Engineer, Citrix R&D India Pvt. Ltd.
2008-Present	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

Research Publications

J. Sridhar and D. K. Panda "Impact of Node Level Caching in MPI Job Launch Mechanisms" *EuroPVM/MPI 2009*, Espoo, Finland, Sep 2009

M. Koop, J. Sridhar and D. K. Panda "TupleQ: Fully-Asynchronous and Zero-Copy MPI over InfiniBand" *IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2009)*, Rome, Italy, May 2009

J. Sridhar, M. Koop, J. Perkins and D. K. Panda "ScELA: Scalable and Extensible Launching Architecture for Clusters" *International Conference on High Performance Computing (HiPC08)*, Bangalore, India, December 2008

M. Koop, J. Sridhar and D. K. Panda "Scalable MPI Design over InfiniBand using eXtended Reliable Connection" *IEEE Int'l Conference on Cluster Computing (Cluster 2008)*, Tsukuba, Japan, September 2008

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. D. K. Panda

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Traditionally computer software has always been written for serial execution. Algorithms were designed with the intention of being implemented as a serial set of instructions on a single processor. During the 1980s, 1990s and the early part of this decade, as application demand for compute cycles increased, the need for additional performance was almost always met by Moore's law [17] – the observation that transistor density in a modern microprocessor doubles every 18 to 24 months. Due to the increasing transistor density, processor designers were able to incorporate more stages and units into a chip and were able to exploit Instruction Level Parallelism (ILP) to derive performance gains.

Increasing transistor density also allowed manufacturers increase clock frequencies of processors which directly led to increased number of instructions being executed per unit time. Thus for a long period of time, increasing clock frequency drove the performance gains in computer software. However, these mechanisms are limited by the power consumption of these CMOS chips which increases with the clock frequency and the transistor density.

The focus has now shifted to exploiting Thread Level Parallelism (TLP) with multi-processors and Data Level Parallelism (DLP) with parallel processing. With frequency scaling also no longer being able to provide increased compute performance, there has been an increased focus on parallel computing where tasks are divided into smaller tasks and computations are performed on multiple processing units. Commodity clusters, which are groups of individual computers connected to each other through an interconnect have become the most popular, cost-effective form of parallel computing. A significant majority of HPC (High Performance Computing) systems are clusters. Several parallel programming models have emerged which utilize clusters. These programming models broadly fit into two categories – shared memory programming models (including distributed shared memory programming) or message passing programming models.

The Message Passing Interface (MPI) [15] is the most widely used message passing Application Programming Interface (API) in distributed memory systems. Message passing programming models such as MPI demand an interconnect technology with low latencies and high bandwidth. InfiniBand Architecture [8] is a high-speed, low latency interconnect primarily used in High Performance Computing (HPC) environments. According to the Top500 [28] list, a biannual list of top supercomputers in the World, the use of InfiniBand has been steadily increasing and as of November 2008, it is being used in 28.2% of the top 500 supercomputers. Table 1.1 shows the adoption rate of InfiniBand [28].

Another consequence of the physical limitations in frequency scaling has been the emergence of multi-core processors which combine two or more independent processor cores into a single package. All the major processor manufacturers have switched to multi-core processors since the early part of this decade. In the High Performance Computing domain, this has lead to an increasing number of multicore processor based

Table 1.1: Adoption of InfiniBand in the Top 500 Supercomputers

Date	IB systems	Percentage	Top Rank
Jun 2005	16	3.2	14
Nov 2005	27	5.4	5
Jun 2006	36	7.2	6
Nov 2006	78	15.6	6
Jun 2007	130	26	8
Nov 2007	121	24.2	3
Jun 2008	121	24.2	1
Nov 2008	141	28.2	1

clusters. In fact, increase in the number of cores per node has been an important factor in the growth of recent clusters. The Sandia Thunderbird [23] cluster introduced in 2006 has 4K nodes – each with dual CPUs for a total of 8K processors. Meanwhile, the TACC Ranger cluster introduced in 2008 has a similar number of nodes at 4K but each with four quad-core CPUs for a total of 64K processors. This trend is poised to continue with Intel recently demonstrating a 80-core processor codenamed polaris.

In the following sections, we provide background information and provide an overview of the Message Passing Interface and the InfiniBand Architecture, arrive at our Problem Statement and discuss our approaches.

1.1 Overview of MPI

The Message Passing Interface (MPI) [15] is a standard developed by the MPI Forum. MPI defines a specific set of API routines that allow processes to communicate between each other. These routines are traditionally used within C, C++ or Fortran programs. Additionally, the API may also be used with any langauage with appropriate binding. Two versions of MPI are in current use. MPI-1 (version 1.2) and MPI-2 (version 2.1) which is a superset of MPI-1 which includes features such as parallel I/O, one-sided operations and dynamic process management.

Conceptually, MPI defines a **Communicator** through which it facilitates both **Point-to-point** and **Collective** communication between processes. MPI-2 also introduces **One-Sided** operations for remote memory access and **Dynamic Process Management** (DPM) to dynamically manage different MPI jobs.

1.1.1 Communicators

MPI Communicators are objects that identify a group of MPI processes. Each process has a unique rank within each communicator and can address each other (for point-to-point communication) with the ranks. Collective operations involve all processes within a communicator. MPI jobs start off with a single communicator `MPI_COMM_WORLD` which includes all processes and can create new communicators with sub groups of processes during the execution of the job.

1.1.2 Point-to-point Communication

Point-to-point communication operations form the basic communication API in MPI. These involve communication between any two processes. For example, at its simplest form, a process may invoke `MPI_Send` to send data to another process which has to make a corresponding call to `MPI_Recv`. MPI also defines non-blocking point-to-point operations which allows MPI applications to overlap computation with communication.

1.1.3 Collective Communication

MPI Collective operations involve all processes within a communicator such as one-to-many and many-to-one operations. All current collective operations are blocking operations. However, there have been recent proposals to introduce non-blocking versions [6]. The MPI standard defines the following set of collective operations.

- `MPI_Bcast` broadcasts data from one process to all other participating processes.
- `MPI_Gater` gathers data from a group of processes
- `MPI_Allgather` gathers data from a group of processes and distributes to all of them
- `MPI_Reduce` combines values from all processes to a single value through specified mathematical operations such as ADD
- `MPI_Allreduce` combines values from all processes and distributes the result to all processes
- `MPI_Scatter` sends data from one process to all other processes
- `MPI_Alltoall` sends data from all processes to to all other processes
- `MPI_Barrier` synchronizes all processes

1.1.4 One-Sided Operations

MPI-2 defines one-sided communication operations that do not need both of the communicating processes to be involved in the operation. These operations – `MPI_Get`, `MPI_Put` and `MPI_Accumulate` facilitate remote memory access. `MPI_Get` reads data from remote memory, `MPI_Put` writes to remote memory while `MPI_Accumulate` performs a reduction operation across other processes. MPI-2 also defines methods to synchronize these one-sided operations. processes.

1.1.5 Dynamic Process Management

The MPI-2 standard introduced Dynamic Process Management (DPM) which allows an MPI process to create new MPI processes or to communicate with other MPI processes that have been started separately. A new set of MPI processes can be created with `MPI_Comm_Spawn`. Connnections between two MPI jobs can be established with `MPI_Comm_accept` and `MPI_Comm_connect`. The `MPI_Comm_join` interface can be used to join two separate MPI jobs. Communication between such separate process groups is facilitated via the intercommunicator that is created by the DPM interfaces.

1.2 Overview of InfiniBand Architecture

The InfiniBand Architecture (IBA)[8] is an industry standard that evolved out of the Virtual Interface Architecture (VIA). It defines a switched fabric offering low latency and high bandwidth - two of the most important properties demanded by modern HPC systems and MPI libraries. Figure 1.1 shows a typical IBA cluster with the switched InfiniBand fabric interconnecting I/O nodes and compute nodes. The compute nodes are connected to the fabric by a Host Channel Adapter (HCA).

1.2.1 Communication Model

InfiniBand provides Operating System bypass to eliminate intermediate copies within the communication stack. It defines a queue based model for interface with the HCAs shown in figure 1.2. A Queue Pair (QP) consists of a send queue and a receive

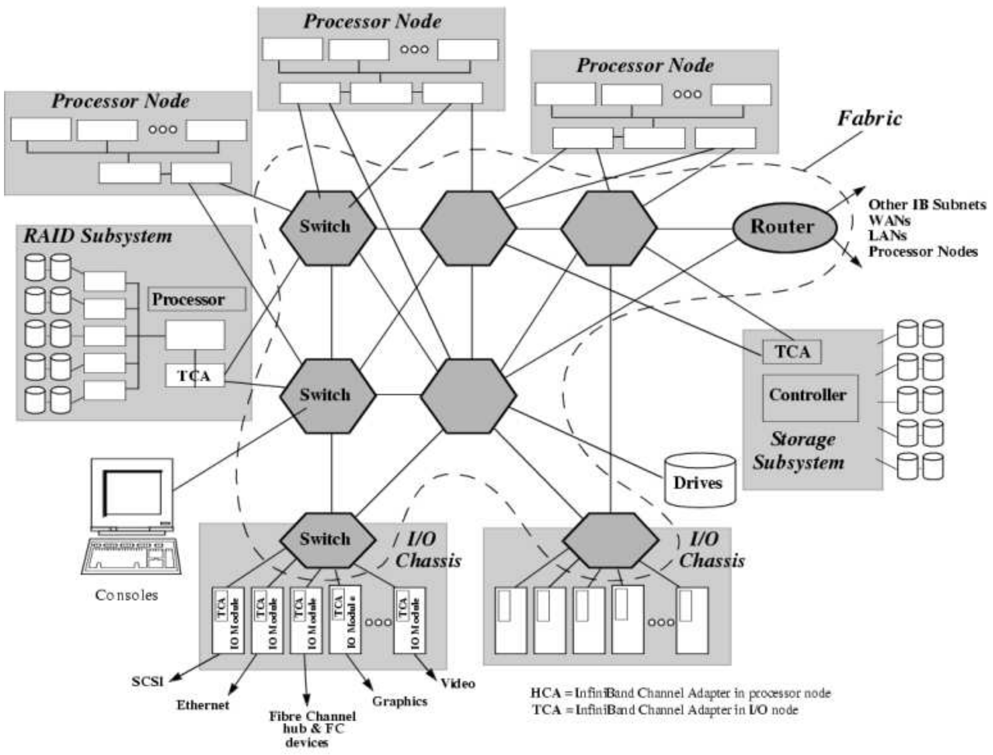


Figure 1.1: The InfiniBand Architecture (Courtesy: The InfiniBand Trade Association [8])

queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication instructions are described in Work Queue Requests (WQR) and are submitted to the work queue. These are now called Work Queue Elements (WQE) and are executed by the Channel Adapters.

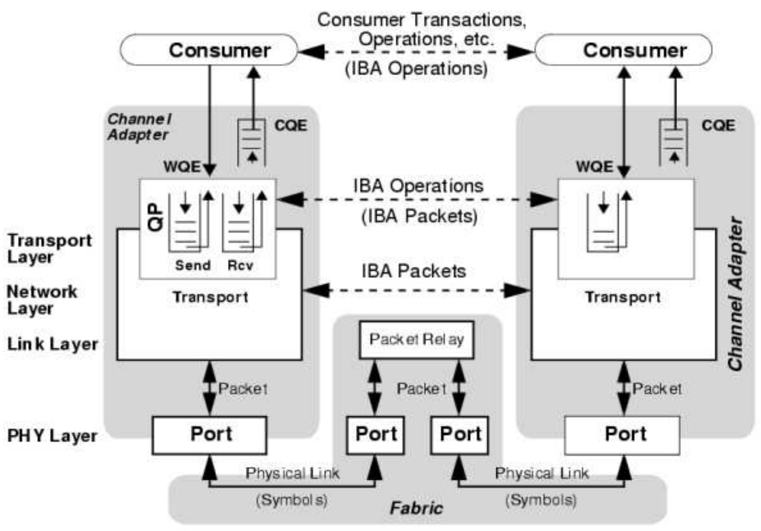


Figure 1.2: The IBA Communication Stack (Courtesy: The InfiniBand Trade Association [8])

The completion of Work Requests is reported through Completion Queues (CQ). Once a WQE is executed, a Completion Queue Entry (CQE) is placed in the corresponding CQ. Completions can be tracked through a callback handler or by polling. Buffers must be posted to a QP to receive messages on that QP and these are consumed in FIFO order.

There are two types of communication semantics in InfiniBand – channel and memory semantics. Channel semantics are equivalent to traditional forms of communication such as sockets where both sides participate in the communication through send and receive operations. Memory semantics are one sided operations in which a process can access the memory of a remote process without the direct participation of the remote process. This is called Remote Direct Memory Access (RDMA). InfiniBand supports both RDMA read and write. Both types of communication semantics require the all data buffers to be registered. This ensures consistency in the HCAs internal address translation and protection tables and also enables the HCA to DMA directly into the destination buffer.

1.2.2 InfiniBand Transports

The InfiniBand specification defines four transport modes – Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC) and Unreliable Datagram (UD). Only RC, UC and UD are required to be supported by InfiniBand HCAs.

Reliable Connection (RC) is the most popular transport service used by MPI implementations over InfiniBand. Since it is a connection-oriented service, a unique QP is required to communicate with each peer. To communicate with N peers, a process needs N QPs. However, the RC transport provides reliable data transmission, RDMA and atomic operations.

Unreliable Datagram (UD) has also been used in some MPI Implementations [10]. It is a connectionless transport and hence a single QP can communicate with any number of peers. However UD is unreliable and any implementation over UD needs to provide software based reliability. Also, packets are limited to MTU of the Interface and the upper layers must also provide fragmentation and reassembly of larger messages. Also, RDMA operations are not possible over UD.

Shared Receive Queues

RC needs a QP per communicating peer. Also, in InfiniBand, it is necessary to pre-post buffers in each QP to handle unexpected receives. These buffers need to be large enough to handle the largest messages that are sent without a prior handshake – via the *eager* protocol in MPI libraries. In typical MPI libraries, this size could be as large as 8KB - 32KB. These bufferes need to be posted on all receive queues. However, they may never be used. This kind of memory usage is not scalable.

To overcome this problem, InfiniBand introduced Shared Receive Queues (SRQ) in version 1.2 of the specification. Instead of having a dedicated RQ per QP, a single SRQ may be used across all QPs in a process. Pre-posted receive buffers can now be shared for messages arriving on any QP. Thus a smaller number of buffers is needed and new buffers can be posted as needed.

Extended Reliable Connection (XRC)

The InfiniBand RC transport is designed to provide connection between two processes. This increases the amount of memory consumed as the number of communicating peers increase. Prior research has shown that a RC QP connection requires several KB of memory and for a 16K job memory usage could reach as high as a gigabyte per process [9].

To address this problem, the XRC transport was introduced. XRC is designed to allow reuse of a connection to a node to reach other processes on a node. Consequently, the number of QPs required grows with total number of nodes rather than total number of process. With the number of cores per node increasing, use of XRC improves the memory usage for QPs considerably.

The services provided by the XRC transport is identical to RC but the connection semantics and addressing is different. When a process is connected to another process on a different node, it can reach all the processes on the peer node via the original connection. However, messages need to be addressed to the SRQ number of the destination process. All the processes on the target node must join a common XRC domain. The HCA can now place data from any QP into any SRQ in the same XRC domain.

1.3 Problem Statement

Clusters continue to increase rapidly in size fuelled by the ever-increasing computing demands of applications. The leading trend in this growth is the increase in the number of cores per node. We examine current scalability challenges in large clusters and evaluate mechanisms to achieve better scalability and performance on modern large scale clusters with tens of thousands of processor cores.

1.3.1 Job Startup

As clusters continue to scale, programming models and their scalability have received a lot of attention in the research community. In addition to these concerns, more basic issues regarding the system software must also be addressed. In particular, the mechanism by which parallel jobs are launched on these large-scale multi-core clusters must be examined. Most parallel programming models require an executable to be launched on each node in the clusters. Many of them such as MPI may require more than one process to be launched per node on multi-core systems. Job launchers also need to facilitate initial communication between all the launched processes to help them discover their peers and initialize their environment. Current job launch mechanisms do not scale to modern large scale clusters. In this light we aim to address the following questions in this thesis:

- **Can we understand the scalability issues and bottlenecks in current job launch mechanisms on large scale multi-core clusters with thousands of nodes?**

- **How can we design a scalable, high-performance job launch mechanism that takes advantage of the emergence of multi-core compute nodes in modern large scale clusters?**

- **Can we use effective caching mechanisms to speed up the job launch process on multi-core processors?**

1.3.2 Point-to-Point Communication

As clusters continue to grow in size, the connection model of modern MPI libraries over InfiniBand does not scale. Each connection takes several hundred KB of memory and the memory usage becomes prohibitively large for clusters with tens of thousands of cores. Most MPI libraries use the Reliable Connection (RC) transport of InfiniBand to establish connections between processes. RC needs a dedicated QP for every communicating peer. This problem was addressed by InfiniBand vendors by introducing the eXtended Reliable Connection (XRC) transport which allows node level connections. We examine if we can use this new transport to greatly reduce number of connections needed during a MPI job.

In this work, we aim to provide answers to the following questions:

- **Can we understand the scalability issues of RC based point-to-point communication in current MPI libraries over InfiniBand?**
- **Can we leverage the increasing use of multi-core processors in large clusters to design a scalable and high-performance point-to-point communication channel?**

1.4 Organization of the Thesis

The rest of this thesis is organized as follows. In Chapter 2 we examine the current job launch mechanisms on large clusters and propose a scalable, extensible and high-performance job launch architecture we call ScELA. In Chapter 3, we study the impact of different caching mechanisms in the launching of MPI jobs and propose a memory efficient caching mechanism to speed up the job launch process. In Chapter 4 we examine scalability challenges in point-to-point communication in MPI libraries over InfiniBand and propose a scalable alternative design over the XRC transport of InfiniBand. We conclude in Chapter 5.

CHAPTER 2

SCALABLE JOB STARTUP IN MULTI-CORE CLUSTERS

2.1 Introduction

Clusters continue to increase rapidly in size, fueled by the ever-increasing computing demands of applications. As an example of this trend we examine the Top500 list [28], a biannual list of the top 500 supercomputers in the World. In 2000 the largest cluster, ASCI White, had $8,192$ cores. By comparison, last year the top-ranked BlueGene/L had over $200,000$ cores. Even as clusters increase in node counts, an emerging trend is increase in number of processing cores per node. For instance, the Sandia Thunderbird [23] cluster introduced in 2006 has $4K$ nodes – each with dual CPUs for a total of $8K$ processors, while the TACC Ranger cluster introduced in 2008 has $4K$ nodes – each with four quad-core CPUs for a total of $64K$ processors.

Programming models and their scalability have been a large focus as cluster size continues to increase. In addition to these concerns, other more basic concerns with regard to the system software must also be addressed. In particular, the mechanism by which jobs are launched on these large-scale clusters must also be examined. All programming models require some executable to be started on each node in the cluster. Others, such as the Message Passing Interface (MPI) [15], may have multiple processes per node – one per core. Our work shows that current designs for launching of MPI jobs can take more than 3 minutes for $10,000$ processes and have trouble scaling beyond that level.

In this work we present a Scalable and Extensible Launching Architecture (ScELA) for clusters to address this need. We note that the initialization phase of most parallel programming models involve some form of communication to discover other processes in a parallel job and exchange initialization information. Our multi-core aware architecture provides two main components: a scalable spawning agent and a set of communication primitives. The spawning agent starts executables on target processors and the communication primitives are used within the executables to communicate necessary initialization information. As redundant information is exchanged on multi-core systems, we design a hierarchical cache to reduce the amount of communication.

To demonstrate the scalability and extensibility of the framework we redesign the launch mechanisms for both MVAPICH [18], a popular MPI library, and the Process Management Interface (PMI), a generic interface used by MPI libraries such as MPICH2 [1] and MVAPICH2 [7]. We show that ScELA is able to improve launch times at large cluster sizes by over 700%. Further, we demonstrate that our proposed framework is also able to scale to at least $32,000$ cores, more than three times the scalability of the previous design.

Although our case studies use MPI, ScELA is agnostic as to the programming model or program being launched. We expect other models such as Unified Parallel C (UPC) [4] to be able to use this architecture as well. In addition, ScELA can be used to run commands remotely on other nodes in parallel, such as simple commands like 'hostname' or maintenance tasks. It is a generic launching framework for large-scale systems.

The remaining parts of this chapter is organized as follows: In Section 2.2 we describe the goals and design issues of our launch framework. We use our framework to redesign two job launch protocols and present these case studies in Section 2.3. Section 2.4 contains a performance evaluation of the ScELA design. Related work is discussed in Section 2.5. We summarize our work in in Section 2.6.

2.2 Proposed Design

In this section we describe the ScELA framework. The main goals of the design are scalability towards a large number of processing cores, ease of extensibility and elimination of bottlenecks such as network congestion and resource limits. For ease of extensibility the various components of ScELA are divided into distinct layers. Figure 2.1 shows an overview of the framework. The following sections describe each of these layers in detail.

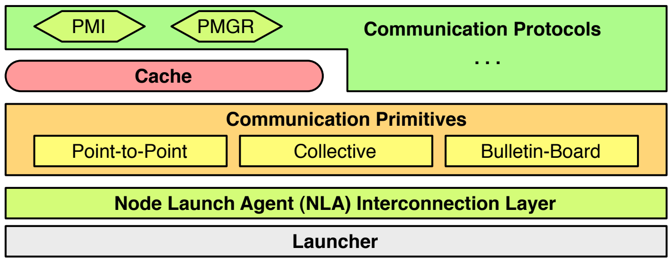


Figure 2.1: ScELA Framework

2.2.1 Launcher

The launcher is the central manager of the framework. The job-launch process starts with the launcher and it is the only layer that has user interaction. The main task of the launcher is to identify target nodes, set up the runtime environment and launch processes on the target nodes.

Process Launching

Modern clusters deploy multi-core compute nodes that enable multiple processes to be launched on a node. On such systems, a launcher would have to duplicate effort to launch multiple processes on a node. ScELA has a Node Launch Agent (NLA) which is used to launch all processes on a particular node. The launcher establishes a connection to target nodes and sets up a NLA on each of them. This mechanism allows the Launcher to make progress on launching processes on other nodes while local NLAs handle node level process launching. The NLAs are active for the duration of the launched process after which they terminate, hence the framework is daemon-less.

Consider a cluster with n compute nodes and c processor cores per node. Table 2.1 shows a comparison of times taken to spawn $n \times c$ processes on such a cluster. T_{conn} is the time taken to establish a connection to a node, T_{launch} is the time taken to spawn a single process and T_{nla} is the time taken to setup a NLA. We see that as the number of cores per node increases, the time taken to start the job decreases with the NLA approach. Since the dominant factor on most clusters is T_{conn} (around 5 ms on our testbed), the use of NLAs on multi-core systems keeps the spawn time practically constant for a fixed number of nodes irrespective of the number of cores per node.

Table 2.1: Time Taken to Spawn Processes With and Without NLAs

With NLAs	Without NLAs
$n \times (T_{conn} + T_{nla}) + c \times T_{launch}$	$(n \times c) \times (T_{conn} + T_{launch})$

Hierarchical Launching

We also design a hierarchical launching scheme for launching NLAs on the target nodes. This is shown in Figure 2.2. The central launcher launches the first NLA. The NLAs then launch the rest of the NLAs in parallel. Figure 2.2 shows the steps involved in launching eight NLAs. We see that the number of steps is reduced to $log(n)$ where n is the number of compute nodes. This scheme helps parallelize the launch phase on clusters with large number of nodes.

Process Health

An important task of job launchers is to handle process termination. When a process fails, a job launcher must clean up other processes. Failure to do so would impact performance of future processes. Having a node level agent allows ScELA to handle monitoring of process health in parallel. The NLAs monitor the health of local processes. When a failure is observed the NLA sends a `PROCESS_FAIL` notification message to the central launcher. The Launcher then sends a `PROCESS_TERMINATE` message to all other NLAs which terminate all processes. User signals such as `SIGKILL` are handled similarly.

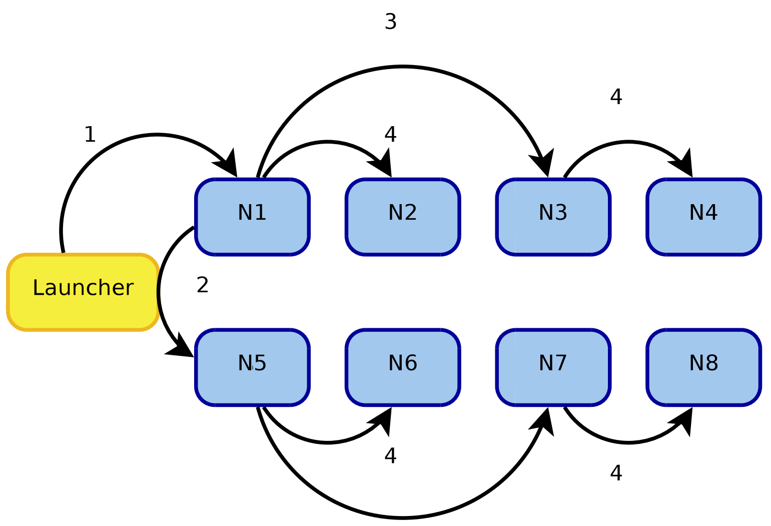


Figure 2.2: Hierarchical Launching

2.2.2 NLA Interconnection Layer

Many programming models require some form of information exchange and synchronization between processes before they complete initialization. For instance, MPI processes may need to discover other processes on the same node to utilize efficient shared memory communication channels or processes may need a barrier synchronization before they can enter a subsequent phase of initialization. Having a connection between every process does not scale for a large number of processes as the number of connections required is $O(n^2)$. Other approaches have all processes connect to a central controller which coordinates information exchange and synchronization. However, when a large number of processes initiate connections to a central controller, it becomes a bottleneck. The resultant network congestion causes TCP SYN packets being dropped. Since SYN retransmission timeouts increase with every attempt on most TCP implementations [25], this introduces a large delay in the overall launch process. In addition, most operating systems limit the number of connections that can be kept open which makes a central controller unfeasible.

We have designed a communication layer over the NLAs to facilitate communication and synchronization between processes. Each NLA aggregates initialization information from all processes on the node. This aggregation limits the total number of network connections needed per entity (process, NLA or the Launcher) on the system. NLAs from different nodes form a hierarchical k-ary tree [16] for communication of information between processes across nodes. The hierarchical tree improves overall parallelism in communication. A k-ary tree allows ScELA to launch processes over an arbitrary number of nodes while also keeping the number of steps required for synchronization and other collective operations such as broadcast or gather at a minimum at $log_k(n)$ where n is the number of nodes. An example of a 3-ary tree of depth 3 is given in Figure 2.3.

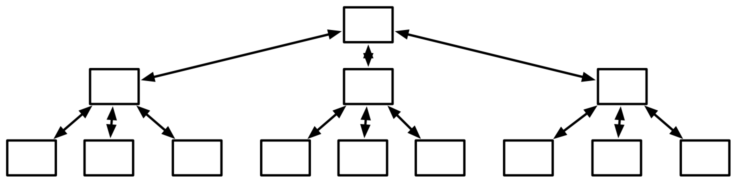


Figure 2.3: Example 3-ary NLA Interconnection (with depth 3)

The degree k of the k-ary tree determines the scalability and the performance of ScELA. An NLA in the hierarchical tree should be able to handle connection setup and communication from all processes on a node as well as the parent and children in the NLA tree. If the degree of the tree is too high, each NLA would have to process too many connections which would create further bottlenecks. If the degree is too low, the depth of the tree would result in too many communication hops.

We determine the degree k dynamically. If n is the number of nodes, we determine an ideal degree k such that the number of levels in the tree, $log_k(n)$, is as follows: $log_k(n) \leq MAX\_DEPTH$. If c is the number of cores per node and $c + k + 1 \leq MAX\_CONN$, then we select k as the degree. If not, we select $k = MAX\_CONN - 1 - c$. The parameter MAX_CONN is the number of connections that an entity can process in parallel without performance degradation. From our experiments (Section 2.4.2) we have determined that a process can handle up to 128 connections with acceptable performance degradation on current generation systems.

2.2.3 Communication Primitives

The characteristics of the information exchange between processes depends on the programming model as well as specific implementations. The communication pattern could be point-to-point, collective communication such as broadcast, reduce, or a protocol such as a global bulletin board. We have designed the following communication primitives over the NLA Interconnection Layer for use by the processes for efficient communication.

Point-to-point Communication Primitives:

Some initialization protocols have processes communicating directly with each other. For such protocols, we have designed two sided point-to-point communication primitives – `NLA_Send` and `NLA_Recv`.

The data from a sender is forwarded to the receiver over the NLA tree. Each process is assigned a unique identifier. During the setup of the NLA Interconnection

Layer, every NLA discovers the location of each process. A process is either on the same node as the NLA, or it can be found in specific lower branch of the NLA tree or higher up the NLA tree.

Collective Communication Primitives:

In most programming models, all processes go through identical initialization phases with identical communication patterns. These communication protocols resemble MPI-style collective communication. To support such protocols, we have designed the following MPI-style collective communication primitives over ScELA.

- `NLA_Gather` – Gather data from all processes to a root process on the root NLA. At each level of the NLA tree, a NLA gathers data from all of its NLA children as well as all processes on its node. Once it has all the data, it forwards the gathered data to its parent NLA.

- `NLA_Broadcast` – Send data from a specified process on the root NLA to all processes. The root NLA sends data down the NLA tree and to all of the processes on the node. On receipt of broadcast data from a parent, each NLA forwards the data down the NLA tree and to all processes on the node.

- `NLA_AllGather` – Gather data from all processes at every process. This primitive is provided as a combination of `NLA_Gather` and `NLA_Broadcast`. The root NLA gathers data from all processes and performs a broadcast operation.

- `NLA_Scatter` – Send specific chunks of data from a process on the root NLA to every process. The root NLA sends data to be scattered down the tree, extracts data meant for processes on its node and sends them to the destination processes. On receipt of a scatter message each NLA forwards it down the NLA tree, extracts data meant for processes on its node and sends them to the destination processes.

- `NLA_AllToAll` – Send specific chunks of data from every processes to every process. The AllToAll primitive is provided as a combination of `NLA_Gather` and `NLA_Scatter`. The root NLA gathers data from all processes, re-organizes the data such that all data destined to a process is grouped together and does a scatter operation.

Bulletin Board Primitives:

Some communication protocols have processes publish information about themselves on a global bulletin board and processes needing that information read it off the bulletin board. To support such protocols over ScELA we have designed two primitives – `NLA_Put` and `NLA_Get`.

`NLA_Put` publishes data to all NLAs up the tree up to the root. When a process needs to read data, it invokes the `NLA_Get` primitive. When data is not available at a NLA, it forwards the request to the parent NLA. When data is found at a higher level NLA, it is sent down the tree to the requesting NLA.

Synchronization Primitive:

In some programming models, the information exchange phase consists of smaller sub-phases with synchronization of the processes at the end of each sub-phase. For instance, in MVAPICH, processes cannot initiate InfiniBand [8] channels until all processes have pre-posted receive buffers on the NIC.

We have designed a synchronization primitive – `NLA_Barrier` which provides barrier synchronization over the NLA tree. Processes are released from an invocation of `NLA_Barrier` primitive only when all other processes have invoked the primitive. The `NLA_Barrier` primitive can be used in conjunction with `NLA_Send` and `NLA_Recv` to design other forms of communication required by a specific communication protocol.

2.2.4 Hierarchical Cache

On multi-core nodes, with communication patterns such as the use of a bulletin board, many processes on a node may request the same information during initialization. To take advantage of such patterns, we have designed a NLA level cache for frequently accessed data. When a process posts information through `NLA_Put`, the data is sent up to the root of the NLA tree while also being cached at intermediate levels. When a process requests information through `NLA_Get`, the request is forwarded up the NLA tree until it is found at a NLA. The response gets cached at all intermediate levels of the tree. Hence subsequent processes requesting the same piece of information are served from a nearer cache. This reduces network traffic and improves the overall responsiveness of the information exchange.

Such a cache is advantageous even on non multi-core nodes or communication patterns without repeated access to common information because the caching mechanism propagates information down the NLA tree. Subsequent requests from other sub-branches of the tree may be served from an intermediate NLA and would not have to go up to the root. In Section 2.3.1 we describe an extension to the `PMI_Put` primitive that enables better utilization of the Hierarchical Cache.

2.2.5 Communication Protocols

As described in Section 2.2.3, the processes being launched may have their own protocol for communicating initialization information. We have designed the ScELA framework to be extensible so that various communication protocols can be developed over it by using the basic communication primitives provided. In Section 2.3 we describe two implementations of such protocols over the ScELA architecture.

2.3 Case Studies

In this section we describe implementations of two startup protocols over ScELA. We first describe an implementation of the Process Management Interface (PMI), an information exchange protocol used by popular MPI libraries such as MPICH2 and MVAPICH2 over the ScELA framework. In addition, we describe an implementation of another startup protocol – PMGR used by MPI libraries such as MVICH [13] and MVAPICH.

2.3.1 Designing the PMI Bulletin Board with ScELA

When MPI processes start up, they invoke `MPI_Init` to set up the parallel environment. This phase involves discovery of other processes in the parallel job and exchange of information. The PMI protocol defines a *bulletin board* mechanism for information exchange. Processes do a `PMI_Put` operation on a `(key, value)` pair to publish information followed by a `PMI_Commit` to make the published information visible to all other processes. When other processes need to read information, they perform a `PMI_Get` operation by specifying a `key`. The PMI protocol also defines a barrier synchronization primitive `PMI_Barrier`.

To implement the PMI bulletin board over the ScELA framework, we utilized the `NLA_Put` and `NLA_Get` primitives. A `PMI_Put` by a process invokes a corresponding `NLA_Put` to propagate information over the NLA tree. When a process does a `PMI_Get`, a corresponding `NLA_Get` is invoked to search for information in the Hierarchical Cache. Since the `PMI_Put`s are propagated immediately, we ignore `PMI_Commit` operations.

We have observed that with the PMI protocol, information reuse is high for some information. In such cases it is beneficial to populate the node level caches even before the first `PMI_Get` request. We have designed an extension to the `NLA_Put` primitive that propagates information to all NLAs in the tree so that all `NLA_Get`s can be served from a local cache. To reduce the number of `NLA_Put`s active in the tree, we aggregate puts from all processes on a node before propagating this information over the tree. When processes invoke `PMI_Barrier`, we invoke the `NLA_Barrier` primitive to synchronize processes. We evaluate our design against the current startup mechanism in MVAPICH2 in Section 2.4.1.

2.3.2 Designing PMGR (Collective Startup) with ScELA

The PMGR protocol defines MPI style collectives for communication of initialization data during `MPI_Init`. These operations also act as implicit synchronization between processes. The PMGR interface defines a set of collective operations – `PMGR_Gather`, `PMGR_Broadcast`, `PMGR_AlltoAll`, `PMGR_AllGather` and `PMGR_Scatter` and an explicit synchronization operation `PMGR_Barrier`.

In our implementation when a process invokes a PMGR primitive, it is directly translated to an invocation of the corresponding collective communication primitive designed over the NLA tree. We evaluate our design against the current startup mechanism in MVAPICH in Section 2.4.2.

2.4 Performance Evaluation

In this section we evaluate the two case studies described in Section 2.3. We evaluate our designs against the previous launching mechanisms in MVAPICH2 and MVAPICH respectively. Our testbed is a 64 node InfiniBand Linux cluster. Each node has dual 2.33 GHz Intel Xeon "Clovertown" quad-core processor for a total of 8 cores per node. The nodes have a Gigabit Ethernet adapter for management traffic such as job launching. We represent cluster size as $n \times c$, where n is the number of nodes and c is the number of cores per node used.

We have written an MPI microbenchmark to measure the time taken to launch MPI processes and time spent in `MPI_Init`, which represents the information exchange phase. For the purpose of these microbenchmark level tests, we disable all optional features that impact job initialization.

2.4.1 PMI over ScELA

In this section, we compare the performance of our implementation of PMI over ScELA (ScELA-PMI) against the default launch framework in MVAPICH2 (MVAPICH2-PMI). The default startup mechanism of MVAPICH2 utilizes a ring of daemons – `MPD` [22] on the target nodes. The launcher – `mpiexec` identifies target nodes and instructs the MPD ring to launch processes on them. PMI information exchange is done over the MPD ring. Figure 2.4 shows the time taken to establish the initial ring. We observe a linear increase which is not scalable over larger number of nodes. We have also observed that the MPD ring cannot be setup on larger sizes such as thousands of nodes. While a MPD ring can be reused for launching subsequent MPI jobs, most job schedulers elect to establish a separate ring as both target nodes and job sizes may be different. Figure 2.5 shows a comparison of the launch times for various system

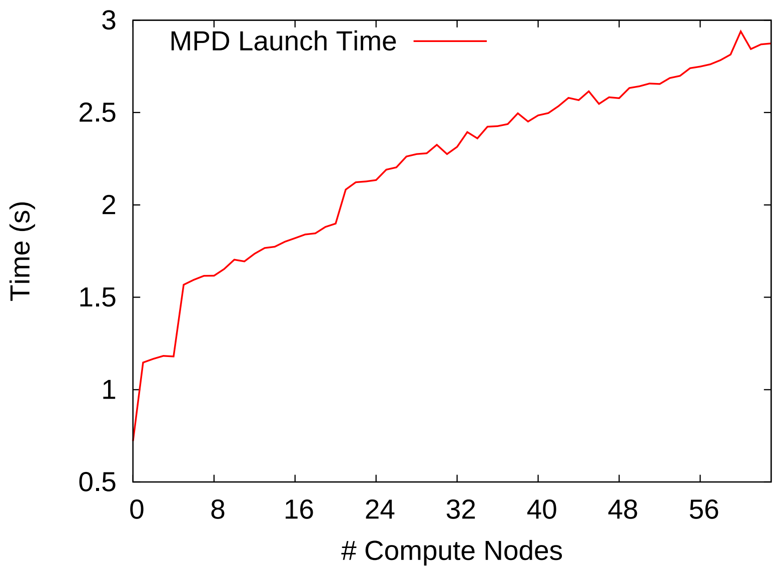
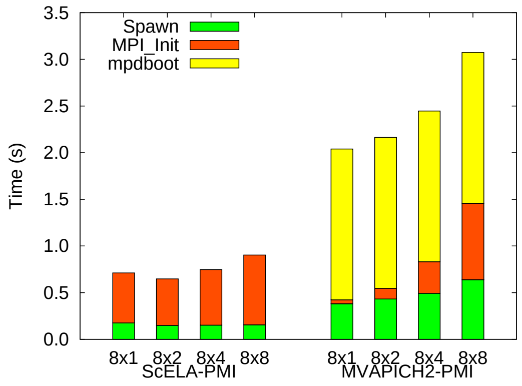
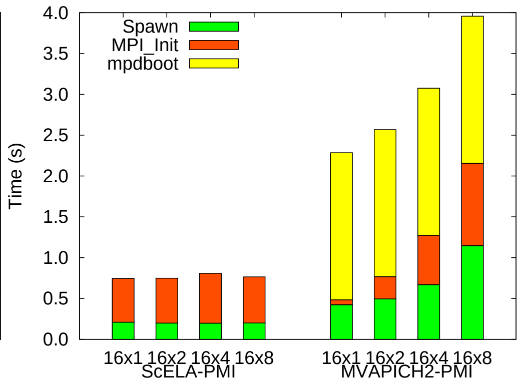


Figure 2.4: Time to Setup MPD Ring with MVAPICH2

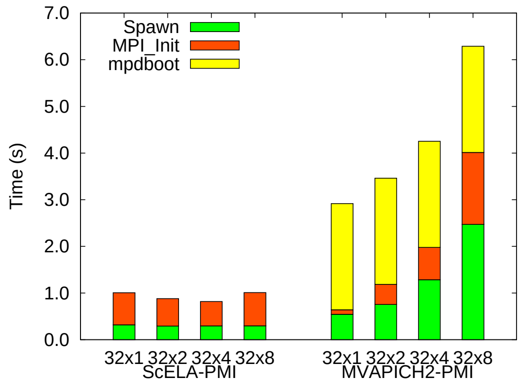
sizes. On ScELA-PMI, the spawn phase represents the time taken for the Launcher to setup NLAs on the target nodes and for the NLAs to launch the MPI processes. The MPI_Init phase represents the time taken to establish the NLA Interconnection Layer and for PMI information exchange. On MVAPICH2-PMI the mpdboot phase represents the time taken to establish the ring of MPD daemons. The spawn phase represents time taken to launch MPI processes over the MPD ring and the MPI_Init phase represents the time taken for information exchange.



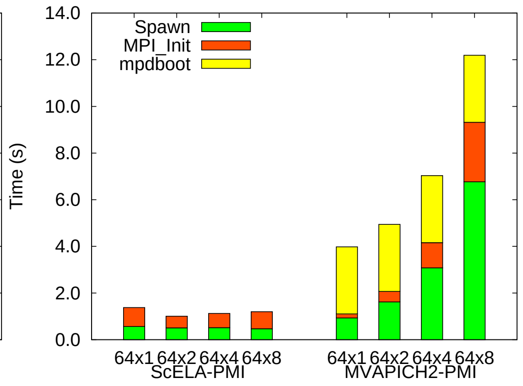
(a) 8 Compute Nodes



(b) 16 Compute Nodes



(c) 32 Compute Nodes



(d) 64 Compute Nodes

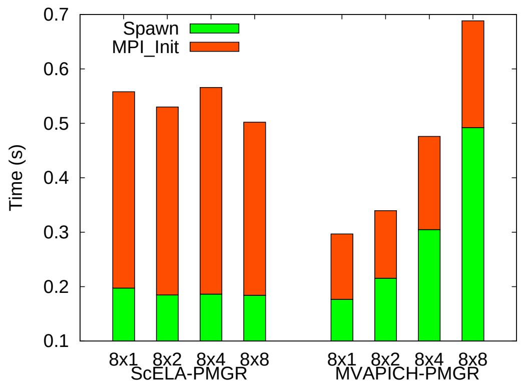
Figure 2.5: Comparison of Startup Time on MVAPICH2

We observe that as we increase the number of processes per node, ScELA-PMI demonstrates better scalability. For a fixed node count, the duration of the spawn phase in ScELA-PMI is constant due to parallelism achieved through NLAs. In Figure 2.5(d) we see the spawn time for MVAPICH2-PMI increase from around 1s to 6.7s when the number of cores used per node is increased from 1 to 8 but ScELA-PMI is able to keep spawn time constant at around 0.5s. At larger job sizes, for instance 512 processes on 64 nodes (64×8 in Figure 2.5(d)), we see an improvement in the MPI_Init phase from around $2.5s$ to $0.7s$ due to the better response times of communication over the NLA Interconnection Layer and due to reduced network communication due to NLA cache hits.

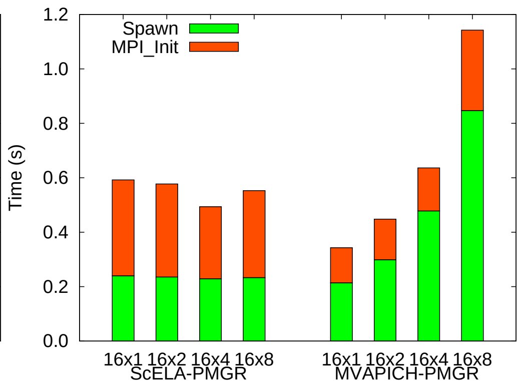
2.4.2 PMGR over ScELA

In this section we compare our design of PMGR over ScELA (ScELA-PMGR) against the default startup mechanism in MVAPICH (MVAPICH-PMGR). The default startup mechanism in MVAPICH has a central launcher that establishes a connection to target nodes and launches each process individually. On multi-core systems, this launcher needs several connections to each node. Also, each MPI process establishes a connection to the central controller which facilitates the PMGR information exchange. As the number of processes increase, this causes a flood of incoming connections at the central controller, which leads to delays due to serialization of handling these requests and network congestion. The number of MPI processes that can be handled simultaneously is also limited by resource constraints such as open file descriptor limits, which is typically 1024.

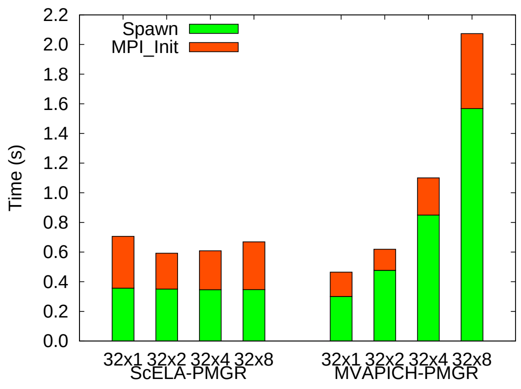
Figure 2.6 shows a comparison of the launch times. With ScELA-PMGR, the spawn phase represents the time taken to setup NLAs on the target nodes and for the NLAs to launch MPI processes on the node. The MPI_Init phase represents the time taken to setup the NLA Interconnection Layer and the PMGR information exchange between MPI processes. With MVAPICH-PMGR, the spawn phase represents the time taken for the central controller to launch each MPI process on target nodes. In the MPI_Init phase, the MPI processes establish connections to the central controller and exchange information over the PMGR protocol. We see that for a fixed node



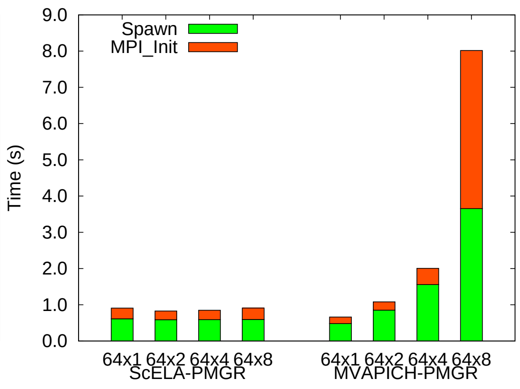
(a) 8 Compute Nodes



(b) 16 Compute Nodes



(c) 32 Compute Nodes



(d) 64 Compute Nodes

Figure 2.6: Comparison of Startup Time on MVAPICH

count, ScELA-PMGR takes constant time for the spawn phase as it benefits from having NLAs while the spawn phase with MVAPICH-PMGR grows with increase in number of processes per node. For instance in 2.6(d), we see that ScELA-PMGR is able to keep spawn time constant at $0.6s$, but on MVAPICH-PMGR the spawn phase increases from $0.5s$ to 3.6 as we increase the number of cores used per node from 1 to 8. When the overall job size is small, the central controller in the MVAPICH startup mechanism is not inundated by a large number of connections. We see that the central controller is able to handle connections from up to 128 processes with little performance degradation in the MPI_Init phase. Hence the MVAPICH startup performs better at a small scale, but as the job sizes increase we observe larger delays in the MPI_Init phase. From Figure 2.6(d) we see that for 512 processes (64×8), the MPI_Init phase takes $4.3s$ on MVAPICH-PMGR, but on ScELA-PMGR it takes around $0.3s$. For 512 processes we see an improvement of 800% in the overall launch time.

Figure 2.7 shows a comparision of ScELA-PMGR and MVAPICH-PMGR on a large scale cluster – the TACC Ranger [27]. The TACC Ranger is an InfiniBand cluster with $3,936$ nodes with four 2.0 GHz Quad-Core AMD "Barcelona" Opteron processors making a total of 16 processing cores per node. The Figure shows the runtime of a simple *hello world* MPI program that initializes the MPI environment and terminates immediately. In terms of number of processing cores, ScELA-PMGR scales up to at least three times more than MVAPICH-PMGR (based on MVAPICH version 0.9.9). On $10,240$ cores, we observe that MVAPICH-PMGR takes around $185s$ while ScELA-PMGR takes around $25s$ which represents a speedup of more than 700%. We also see that MVAPICH-PMGR is unable to scale beyond $10,240$ cores, while ScELA-PMGR is able to scale to at least 3 times that number.

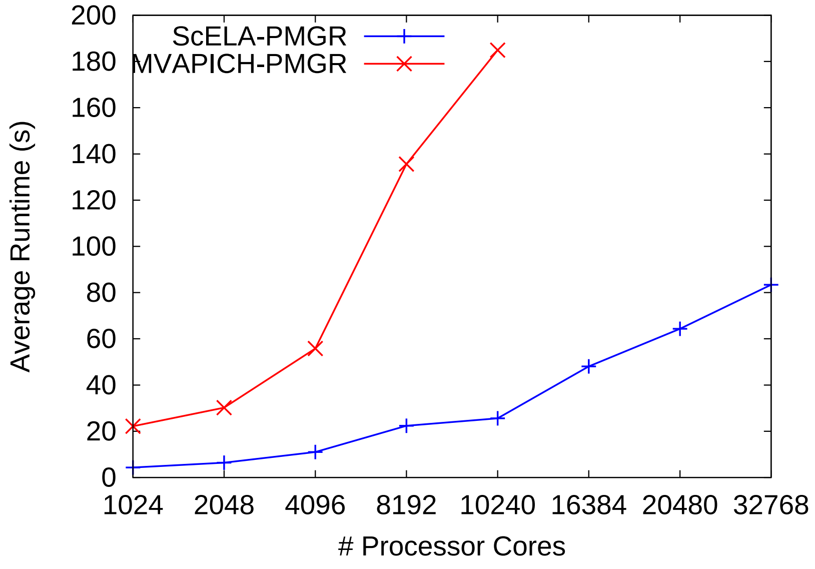


Figure 2.7: Runtime of Hello World Program on a Large Scale Cluster (Courtesy TACC)

2.5 Related Work

The scalability and performance of job startup mechanisms in clusters have been studied in depth before. Yu, et. al. [29] have previously explored reducing the volume of data exchanged during initialization of MPI programs in InfiniBand clusters.

In our work, we have assumed availability of executable files on target nodes through network based storage as this is a common model on modern clusters. Brightwell, et. al. [3] have proposed a job-startup mechanism where network storage is not available.

SLURM [14] is a resource manager for Linux clusters that implements various interfaces such as PMI and PMGR for starting and monitoring parallel jobs. Unlike

ScELA, SLURM has persistent daemons on all nodes through which it starts and monitors processes.

2.6 Summary

In this work we study the scalability challenges in job launching in large clusters. We identify the problems in current mechanism and propose a Scalable and Extensible Launching Architecture (ScELA) that scales to modern large scale clusters such as the 64K processor TACC ranger.

With an implementation of our architecture, we have achieved a speedup of 700% in MPI job launch time on a very large scale cluster at $10,240$ processing cores by taking advantage of multi-core nodes. We have demonstrated scalability up to at least $32,768$ cores.

CHAPTER 3

IMPACT OF NODE LEVEL CACHING IN MPI JOB LAUNCH MECHANISMS

3.1 Introduction

In this work, we continue the study on the job launch phase with partticular focus on MPI job launching to analyze the communication between processes during job initialization with the goal to improve the startup performance. We work on the ScELA framework introduced in Chapter 2. In particular we aim to study the performance benefits of caching information at the node level on multi-core processors. We propose four design alternatives for caching mechanisms that improve the performance of the startup phase in MPI applications. The first three designs include: Hierarchical Cache Simple (HCS), Hierarchical Cache with Message Aggregation (HCMA) and Hierarchical Cache with Message Aggregation and Broadcast (HCMAB). We study the communication pattern during a typical MPI job startup phase and enhance our designs to introduce the Hierarchical Cache with Message Aggregation, Broadcast and LRU (HCMAB-LRU) which is memory efficient while retaining the performance benefits of earlier methods. We evaluate our designs on a 512 core InfiniBand cluster to demonstrate the performance benefits of pre-populating node level caches. This reduces the time taken for a typical information stage during startup to less than one tenth while adhering to an upper bound on the memory used for the cache.

The rest of this work is organized as follows. We study the impact of node level caching and design alternatives in Section 3.2. We evaluate our designs in Section 3.3. We conclude and give future directions in Section 3.4

3.2 Proposed Designs

In this Section we describe alternative design mechanisms for caches over the hierarchical NLA network.

3.2.1 Hierarchical Cache Simple (HCS)

Figure 3.1 shows the basic caching mechanism in ScELA – the Hierarchical Cache Simple (HCS). When an MPI application publishes information with the `PMI_Put` operation, the data is sent to the local NLA. The NLA adds it to its local cache and forwards the data to the upper level NLA. The upper level NLAs add the information to their respective caches and forward it up the NLA tree.

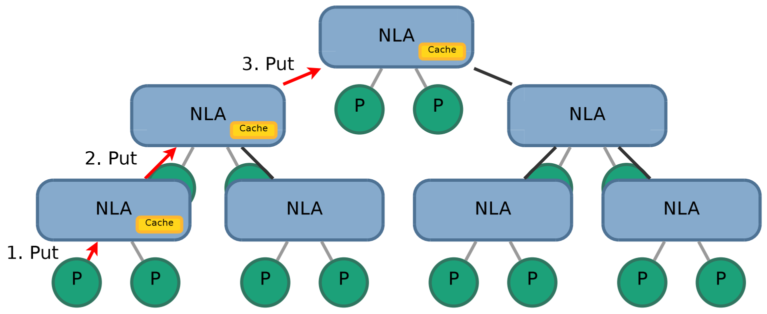


Figure 3.1: Hierarchical Cache Simple

When an MPI application requests for information via the `PMI_Get` operation, the local NLA checks if the data is available in its cache. If so, it responds with the data. If not, the request is forwarded up the NLA tree until an NLA finds the information in its cache. The response is cached in all intermediate NLAs as it travels down the tree. Since all `(key, value)` pairs get cached in the root of the NLA tree, worst case memory usage is $O(p \times n)$ where p is the number of processes and n is the number of `(key, value)` pairs published by each process.

3.2.2 Hierarchical Cache with Message Aggregation (HCMA)

In HCS, the number of messages sent over the NLA tree is large. The root of the NLA tree receives $O(p \times n)$ individual messages each containing a `(key, value)` pair. Sending a large number of small messages over the NLA tree is a costly operation. We observe that all MPI libraries ensure that MPI processes are synchronous when publish and retrive information. The typical information exchange phase can be summarized by the following pseudocode:

```
PMI_Put (mykey, myvalue);
PMI_Barrier ();
...
val1 = PMI_Get (key1);
val2 = PMI_Get (key2);
...
```

Since we know that all MPI processes on a node publish information around the same time, we can aggregate this information at the local NLA before forwarding it up the tree. The higher level NLAs wait for all local processes to publish information as well as aggregate information from all lower level NLAs and send a single aggregate message up the NLA tree after populating the local cache. Figure 3.2 shows such a mechanism. We call this mechanism as Hierarchical Cache with Message Aggregation (HCMA). With the message aggregation, we reduce the number of messages sent over the NLA tree. The root of the NLA tree receives $O(log_k(p) \times n)$ messages (where k is the degree of the NLA tree) since it only receives one message from each NLA and smaller messages from the local MPI processes. The worst case memory usage is the same as HCS at $O(p \times n)$ which is related to the amount of data cached at the root NLA.

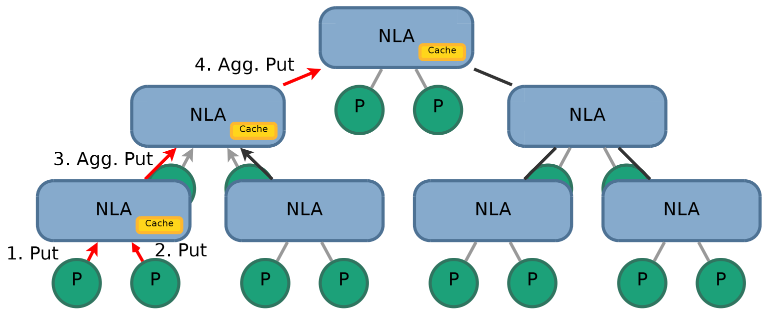


Figure 3.2: Hierarchical Cache with Message Aggregation

The handling of information retrieval in HCMA is identical to HCS.

3.2.3 Hierarchical Cache with Message Aggregation and Broadcast (HCMAB)

In both HCS and HCMA, when a MPI process requests for information that is not available in the local NLA cache, the request is forwarded up the NLA tree. The root NLA has data from all MPI processes. Thus the number of `PMI_Get` messages reaching the root is high and the responses need to travel multiple hops before the MPI application receives the data.

On multi-core clusters, we observe that due to the presence of multiple MPI processes on each compute node, most of the data would eventually be requested by some MPI process on the node. Thus, pre-populating all NLA caches would be beneficial in terms of reducing the number of messages sent over the network. This method – Hierarchical Cache with Message Aggregation and Broadcast (HCMAB) is similar to HCMA with one additional step. After the root NLA has accumulated data from all processes, it broadcasts an aggregate message to all NLAs. This ensures that all `PMI_Get` requests are served from local NLA caches.

In this method, each NLA in the tree has memory requirement of the order of $O(p \times n)$ since all information is cached in all NLAs.

3.2.4 Hierarchical Cache with Message Aggregation, Broadcast with LRU (HCMAB-LRU)

Since HCMAB has the least number of messages travelling over the broadcast network and had all data requests served from a local cache, it offers the best performance of the three prior schemes. However populating every NLA cache involves memory usage of $O(p \times n)$ on every NLA where p is the number of MPI processes and n is the number of $(key, value)$ pairs published by every MPI process. Job launchers however need to keep their memory usage low. We observe that the information exchange during `MPI_Init` happens in stages where ranks publish information and retrieve information published by other ranks. The exchange is repeated in subsequent stages with new information. Thus we can limit memory used for the cache to $O(p)$ such that the cache can hold information required for one particular stage of information exchange. We use the Least Recently Used (LRU) cache replacement algorithm when storing data in the cache so that information from prior stages are discarded. We call this mechanism the Hierarchical Cache with Message Aggregation, Broadcast and LRU (HCMAB-LRU).

3.2.5 Comparison of Memory Usage

Table 3.1 shows a comparison of the memory usage between the four caching mechanisms for n rounds of information exchange. We observe that HCMAB-LRU is able to maintain a strict upper bound on its memory usage while the other schemes use increasing amount of memory as the amount of information exchanged increases.

Table 3.1: Node Level Memory Usage of Proposed Caching Mechanisms on Various Cluster Sizes for n published $(key, value)$ pairs

MPI Job Size (p)	Caching Mechanism			
	HCS	HCMA	HCMAB	HCMAB-LRU
64	$O(64 \times n)$	$O(64 \times n)$	$O(64 \times n)$	$O(64)$
256	$O(256 \times n)$	$O(256 \times n)$	$O(256 \times n)$	$O(256)$
1024	$O(1024 \times n)$	$O(1024 \times n)$	$O(1024 \times n)$	$O(1024)$
4096	$O(4096 \times n)$	$O(4096 \times n)$	$O(4096 \times n)$	$O(4096)$
16384	$O(16384 \times n)$	$O(16384 \times n)$	$O(16384 \times n)$	$O(16384)$
65536	$O(65536 \times n)$	$O(65536 \times n)$	$O(65536 \times n)$	$O(65536)$

3.3 Performance Evaluation

In this Section we evaluate the four design alternatives presented in Section 3.2. Our testbed is a 64 node InfiniBand Linux cluster. Each node has dual 2.33 GHz

Intel Xeon "Clovertown" quad-core processor for a total of 8 cores per node for a total of 512 processor cores. The nodes have one Gigabit Ethernet adapter each for management traffic such as job launching. We represent cluster size as $n \times c$, where n is the number of nodes and c is the number of cores per node used.

We implement our design alternatives over the ScELA-PMI startup mechanism available in MVAPICH2 a popular MPI library for InfiniBand and iWarp that uses PMI during the job launch phase.

3.3.1 Simple PMI (1:2) Exchange

We profile a part of the code where MVAPICH2 processes exchange information with two peers to form a ring connection over InfiniBand. In this phase each MPI process publishes one `(key, value)` pair using `PMI_Put` and retrieves `values` published by two other MPI processes. For instance, this kind of information exchange is used in MVAPICH2 to establish a ring network over InfiniBand. Figure 3.3 shows the performance of the caching mechanims.

We see that HCS performs the worst since the number of messages travelling the NLA tree during both the publishing phase and the retrieval phase is highest. HCMA performs better since it reduces the number of messages travelling over the network during the publishing phase through message aggregation. HCMAB and HCMAB-LRU perform virtually identically since the communication pattern is identical and the reduce the time taken for the phase to less than one third of HCS. Note that HCMAB-LRU uses much less memory since it does not retain information from previous stages of communication in the NLA caches.

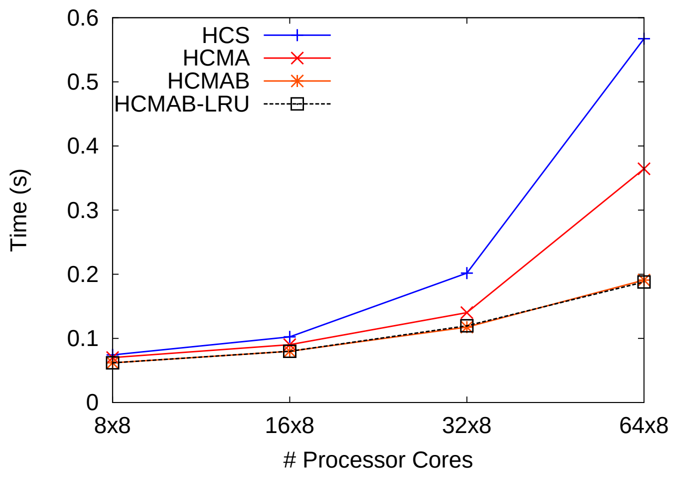


Figure 3.3: Time Taken for a Simple PMI Operation with One Put and Two Gets

3.3.2 Heavy PMI (1:p) Exchange

In another common form of information exchange, each MPI process publishes one piece of information. All p MPI processes then read information published by every other MPI process. For instance, this method is used to learn the InfiniBand LIDs of all MPI processes in MVAPICH2. Figure 3.4 shows the results from profiling such an exchange. We observe the performance benefits of caching mechanisms such as HCMAB and HCMAB-LRU that populate all the caches prior to any `PMI_Get` requests from MPI processes. We observe that these mechanisms reduce the time taken for such exchanges to one tenth that of HCS. In these methods all data is served from the local NLA cache and the number of messages travelling through the NLA tree is minimal. These two factors improve the performance of the information exchange greatly. As the number of messages exchanged is low, these mechanisms can scale to clusters of larger node counts.

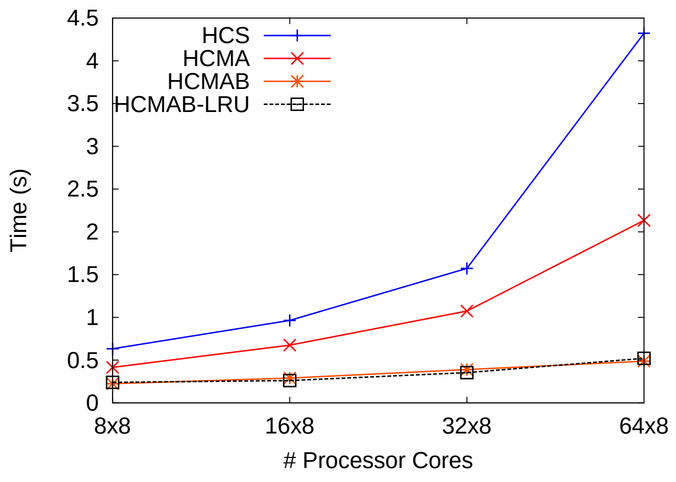


Figure 3.4: Time Taken for a PMI Operation with One Put and p Gets

3.4 Summary

In this work we have proposed four alternatives for node level caches in MPI job launchers. The simplest method – HCS improves the scalability and performance of the startup while keeping the average memory usage per node low in the job launcher. We improve the performance using message aggregation in HCMA. We propose an enhancement over this method that takes advantage of communication patterns used by typical MPI libraries that use PMI with HCMAB. We propose an enhancement over HCMAB to cap memory used to a fixed value. We reduce the performance of communication phases to around a tenth with our optimizations. Though we discuss our designs in terms of the k-nomial tree based ScELA framework, similar reasoning applies to caching information other startup mechanisms such as the ring based MPD in MPICH2 [1].

In the future, we propose to study node level caches and other optimizations over much larger clusters. We plan to evaluate distributed caching to reduce memory usage further in the presence of hundreds of thousands of MPI processes.

CHAPTER 4

SCALABLE POINT-TO-POINT COMMUNICATION IN MULTI-CORE INFINIBAND CLUSTERS

4.1 Introduction

As clusters continue to grow in size, the scalability of point-to-point communication channels in MPI libraries are starting to become a bottleneck. In this work, we study the scalability challenges in point-to-point communication channels in MPI libraries over InfiniBand and propose a highly scalable alternative channel over the eXtended Reliable Connection (XRC) transport of InfiniBand.

4.2 Overview of Point-to-Point Communication in MPI Libraries

Point-to-point communication is the most important aspect of MPI libraries. All higher level operations such as collectives are built upon point-to-point connections. Any MPI implementation must provide scalable and high-performance mechanisms for point-to-point communication between any two MPI processes.

In this chapter we study the scalability challenges in designing efficient, scalable and high-performance communication in MPI libraries over InfiniBand libraries.

4.2.1 Scalability Challenges

Current MPI implementations over InfiniBand use the RC as their primary transport. Prior work has shown that the RC transport requires several kilo bytes of memory per connection [9]. This could lead to up to a gigabyte of memory being used up by just InfiniBand connections for large scale jobs such as those running at 16K processes.

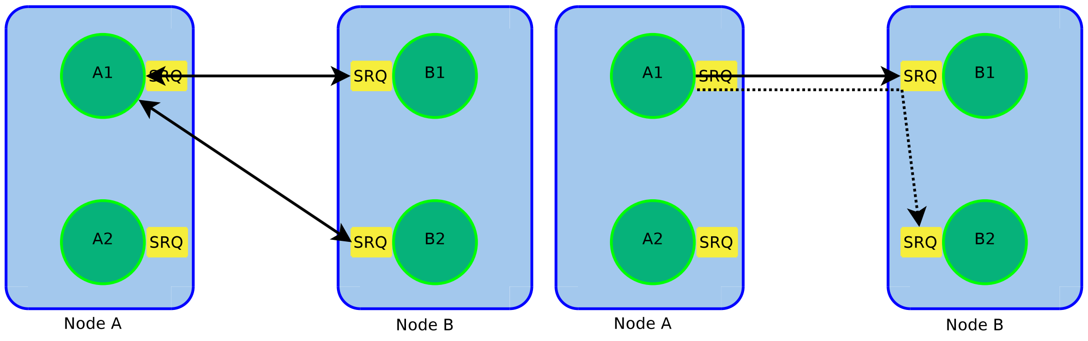
Some MPI libraries such as MVAPICH [19] use and adaptive connection management scheme to reduce the number of connections created [30]. Unlike the static connection management scheme where connections to all peers are created at MPI_Init, in adaptive connection management, connections are established on-demand, i.e., an IB connection is established only when a process needs to communicate with another process. This reduces the number of connections created by a process if it does not talk to every other process during the duration of the job. This mechanism is useful when MPI processes form cliques and communicate within these. For example in cases when MPI processes only communicate with their nearest neighbours. However, in this adaptive method, if a job is eventually fully connected, the number of connections created and the memory used is the same as the static scheme.

MPI libraries also use shared memory based communication [5] for intra-node peers to reduce the number of network connections. The large number of connections also leads to cache pollution in the InfiniBand HCA cache. Many other research work has focused on the scalability of MPI libraries over InfiniBand. Some of these reduce the consumption of buffers required by InfiniBand connections by using the SRQ mechanism in recent InfiniBand versions [24], [26]. Message coalescing has been used to reduce the memory usage [9]. An adaptive approach where the UD transport of InfiniBand is used for the first sixteen messages and then switches to RC has also been proposed [30]. Connection-less UD based designs [11] and zero-copy over UD [12] have been recently evaluated. A hybrid approach that switches between UD and RC based on the application communication pattern and frequency was presented in [10].

In our work, we design a MPI communication channel over the eXtended Reliable Connection (XRC) transport of InfiniBand available with recent InfiniBand adapters. As described in Section 1.3.2, XRC enables connections to be established to nodes rather than processes. This is particularly relevant with the advent of multi-core processors. This means that a process on node A communicating with another process on node B over XRC can reach all processes on node B through the original connection. The rest of this chapter is organized as follows. In the Section 4.3 we present our design for MPI over XRC, we evaluate our design in Section 4.4 and we summarize in Section 4.5.

4.3 Proposed Design

The XRC transport of InfiniBand allows a sender to reuse an existing connection to a process on a node to reach any other process on the same destination node. For this mechanism to be allowed by the HCA, all processes on a node need to join an XRC domain. Once all processes have joined a common domain, the HCA can place data arriving on any QP tied to the domain into a SRQ belonging to any process in the same domain. Thus XRC connections connect a process to a node. A process can address any process on a target node by specifing the SRQ number in the WRE (Work Request Entry). A comparison of RC and XRC is shown in Figure 4.1. In the example shown, all nodes are dual core. Process A1 on node A needs two distinct RC connections to process B1 and B2 on node B. However when XRC transport is used a single connection can be used to reach both B1 and B2.



(a) RC Connections (b) XRC Connections

Figure 4.1: Comparison of RC and XRC InfiniBand Transports

4.3.1 MPI over eXtended Reliable Connection (XRC)

We use the XRC transport to design an on-demand connection management in MVAPICH and MVAPICH2 [19]. InfiniBand connections are created to a peer only when there is a need to transfer some data, i.e., connections are established when the first data packet is sent to a peer. We use a hash table to keep track of existing connections to all nodes. When a message is to be sent to a new peer process to which we do not have a connection, we check the hash table to see if we already have another XRC connection that can be reused. If so, we reuse the existing connection to communicate with this new peer and mark the connection as an indirect connection. This design can be described by the pseudo-code in Figure 4.2.

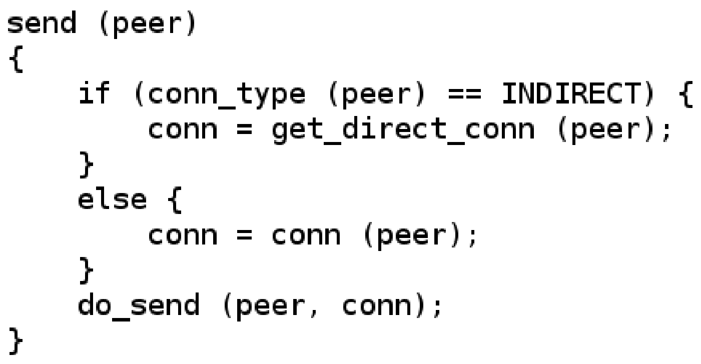
```
connect (peer)
{
    dest_host = host(peer);
    conn = lookup_connection_hash (dest_host);
    if (conn) {
        reuse_connection (peer, conn);
    }
    else {
        conn = do_connect (peer);
        add_connection_hash (conn);
    }

}
```

Figure 4.2: XRC Connection Management

All processes create UD QPs to enable connection setup handshake messages. If we do not find a connection to a node, we send a UD message to the peer requesting a new connection and mark the connection as a direct connection. In either case, the remote SRQ number is obtained through UD messages. This SRQ number is used in all WREs when sending data to this peer. When sending data to a peer, we need to find the original XRC Queue Pair if it is a indirect connection. Once we find the original Queue Pair, we populate the destination SRQ in the WRE and post a send to the HCA. The HCA on the destination node places the data in the defined SRQ at the receiver if the domain is the same. The sending mechanism can be described by the pseudo-code in Figure 4.3.

Through the use of XRC, in the ideal case, we can reduce the number of connections needed per process to the number of nodes being used rather than the number of processes (cores). Table 4.1 shows a comparison of the number of connections created by each process in a fully connected MPI job. We observe that as the number of



```
send (peer)
{
    if (conn_type (peer) == INDIRECT) {
        conn = get_direct_conn (peer);
    }
    else {
        conn = conn (peer);
    }
    do_send (peer, conn);
}
```

Figure 4.3: Sending over XRC

cores per node increases, the number of connections created in case of XRC reduces drastically compared to RC.

Table 4.1: Comparison of the Number of Connections in RC vs XRC

Transport	Nodes	Cores	Job Size	Connections per Process
RC	n	c	$n \times c$	$n \times c - 1$
XRC	n	c	$n \times c$	n

4.3.2 Dynamic Process Management

MVAPICH2 which is a MPI-2 implementation over InfiniBand supports the Dynamic Process Management feature of MPI-2. With DPM, a group of MPI processes can spawn another group of MPI processes. DPM also allows a group of MPI processes to establish connections to another group of MPI processes. When a MPI job spawns another MPI job, the processes in the second job may be spawned on the same nodes as the original job. In such scenarios, if a process from a process group

X needs to talk to a process from process group Y and already has a connection to the destination node, we take advantage of XRC to reuse the same connection. Such a scenario is shown in Figure 4.4. In this example, we use process A4's connection to process B4 to talk to a process B5 in another process group.

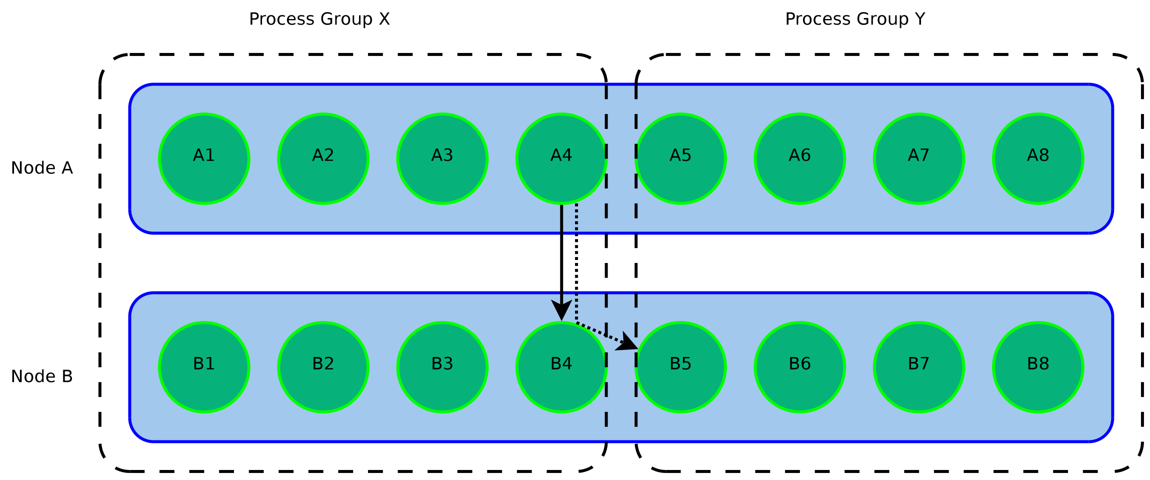


Figure 4.4: MPI Dynamic Process Management with XRC

In a MPI job without DPM, all processes start and terminate around the same time. However, when DPM is being used, a communicating peer may terminate at any time. If we have a direct connection to a peer that terminates, we lose connectivity to all peers who were reachable through this connection. To overcome this problem, we use a special XRC receive only QP on the receive side of a XRC connection. When a direct connection is established, the receiver side creates a special receive only QP. Other peers who're reachable through this QP also register to this QP as needed. When a receiver process no longer needs this connection, it unregisters itself from this QP. The kernel only deletes the receive only QP when no process is registered to it. Hence even if the original receiver process terminates, the other processes on the node can still receive data on this XRC QP.

Through the use of XRC, we reduce the number of connections created per process. In large MPI jobs on multi-core processors, this not only reduces memory used, but also reduces cache pollution in the HCA. In the next section we present a performance evaluation of our designs.

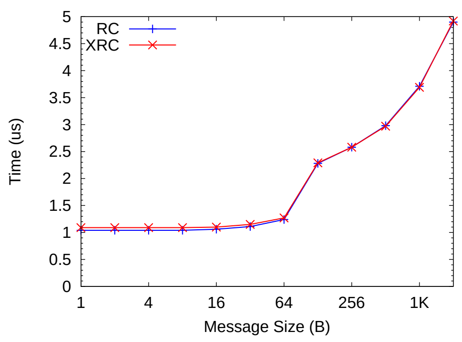
4.4 Performance Evaluation

In this section, we present an evaluation of our deisgn. We have implemented our design on both MVAPICH and MVAPICH2 and we compare our design to the standard RC based designs. We start our evaluation with MPI level microbenchmarks [20]. We then use more comprehensive benchmarks, the NAS Parallel Benchmarks [2] to show the impact of the XRC based MPI design.

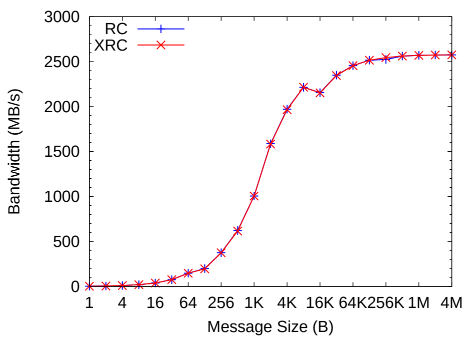
4.4.1 MPI Microbenchmarks

We evaluate our design on two Linux nodes equipped with Mellanox ConnectX InfiniBand HCAs connected through a switch. Each of these nodes have the Intel Xeon “Harpertown” processor and have a dual quad-core processor. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [21] version 1.4.

Figure 4.5 shows the results of the MPI level micro-benchmark evaluation. Due to the extra processing involved with each send operation, there is a small penalty in small message latency as can be seen in Figure 4.5(a). However, the bandwidth achieved is practically identical to the RC based designs as seen in Figure 4.5(b).



(a) MPI Latency (1 pair)



(b) MPI Bandwidth (1 pair)

Figure 4.5: MPI Microbenchmark Evaluation of MVAPICH with RC and XRC

4.4.2 Application Benchmarks

We use the NAS Parallel Benchmarks (NPB) [2] to compare the RC and the XRC based MPI designs. NPB is developed at NASA and is a set of benchmarks which are derived from the computation kernels of common Computational Fluid Dynamics (CFD) applications. We run our experiments on a 8 node InfiniBand Linux cluster. Each node has a dual 2.33 GHz Intel Zeon "Clovertown" quad-core processor for a total of 8 cores per node. We use the NPB class B benchmark size for our evaluation.

The results of the comparison is shown in Table 4.2. For each benchmark we show the average, minimum and maximum number of InfiniBand QPs created per process in addition to the running time. An examination of the running time shows that there is practically little difference between the RC and the XRC designs. However, the number of connections created per process is markedly fewer with XRC. FT and IS which create all-to-all connectoins highlight this trend in particular.

Table 4.2: Comparison of the Number of Connections in RC vs XRC with NPB (Class B)

App	Transport	Number of QPs per Process			Running Time
		Average	Minimum	Maximum	
BT	RC	6.06	6	7	26.63
	XRC	4.06	4	5	26.60
CG	RC	3.84	3	4	18.14
	XRC	3.5	3	4	18.04
EP	RC	3	3	3	3.28
	XRC	3	3	3	3.24
FT	RC	55.91	54	56	8.57
	XRC	6.95	5	7	8.53
IS	RC	53.16	50	56	0.44
	XRC	6.98	6	7	0.42
LU	RC	3.81	3	5	17.81
	XRC	3.81	3	5	17.75
MG	RC	4.98	4	6	1.75
	XRC	4	4	4	1.75
SP	RC	6.06	6	7	46.57
	XRC	3.73	3	5	46.54

4.5 Summary

With this work, we examin the scalability constraints in RC based communication channels in MPI libraries over InfiniBand. We propose a design based on eXtended Reliable Connection (XRC) that performs identical to the RC based design with reduced memory usage that allows it to scale to larger clusters.

Our design also takes advantage of the XRC paradigm to reuse connections even when two different MPI jobs talk to each other through Dynamic Process Management (DPM).

Implementations of our designs are available in MVAPICH version 1.1 and the upcoming version 1.4 of MVAPICH2, two popular MPI libraries over InfiniBand.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Summary

Clusters continue to scale in core counts. Node counts are increasing significantly, but much of the growth in core counts is coming from multi-core clusters. In our work we have demonstrated a scalable launching architecture that improves the launch performance on multi-core clusters by more than an order of magnitude than previous solutions. Although our case studies have been with two MPI libraries, we have presented an architecture extensible to any cluster launching requirements. For launching parallel jobs, we provide scalable and efficient communication primitives for job initialization. With an implementation of our architecture, we have achieved a speedup of 700% in MPI job launch time on a very large scale cluster at $10,240$ processing cores by taking advantage of multi-core nodes. We have demonstrated scalability up to at least $32,768$ cores. These solutions are being used by several large scale clusters running MVAPICH such as the TACC Ranger – currently the largest InfiniBand based computing system for open research.

We have examined the communication pattern in the job launche phase and various opportunities to use node level caching to speed up the job launch phase. We have proposed four alternative mechanisms for node level caches and have studied the scalability aspects and memory usage of each of them. The simplest method – HCS improves the scalability and performance of the startup while keeping the average memory usage per node low in the job launcher. We improve the performance using message aggregation in HCMA. We propose an enhancement over this method that takes advantage of communication patterns used by typical MPI libraries that use PMI with HCMAB. We propose an enhancement over HCMAB to cap memory used to a fixed value. We reduce the performance of communication phases to around a tenth with our optimizations. Though we discuss our designs in terms of the k-nomial tree based ScELA framework, similar reasoning applies to caching information other startup mechanisms such as the ring based MPD in MPICH2 [1].

We have also examined the scalability constraints with current RC based point-to-point communication channels in InfiniBand MPI libraries and proposed a scalable comunication channel over the eXtended Reliable Connection (XRC) InfiniBand transport available in recent IB adapters with reduced memory usage. Our design shows a marked decrease in the number of connections created in multi-core clusters.

5.2 Future Work

With the recent demonstration of a 80 core processor by Intel, the number of cores per node on large scale clusters is projected to increase further. In our job launch design, we can use more efficient communication channels such as UDP or shared memory for communication between processes and the NLA on a node so that the degree of the NLA tree can be decoupled from the number of cores on a node.

We also plan to evaluate the use of eXtended Reliable Connection (XRC) transport of InfiniBand in collective intensive communication and understand the behaviour, identify potential bottlenecks and optimization.

5.3 Software Distribution

The ScELA design is availabe in MVAPICH version 1.0 and MVAPICH2 version 1.2 onwards. Implementations of HCS and HCMAB caching mechanisms are integrated into the 1.2 release of MVAPICH2. We plan to integrate HCMAB-LRU into an upcoming release of MVAPICH2. Our scalable MPI channel over XRC InfiniBand transport is available in MVAPICH version 1.1 and the upcoming version 1.4 of MVAPICH2.

MVAPICH and MVAPICH2 are popular MPI libraries over InfiniBand used by over 900 organizations around the World.

BIBLIOGRAPHY

[1] Argonne National Laboratory. MPICH2 : High-performance and Widely Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. volume 5, pages 63–73, Fall 1991.

[3] R. Brightwell and L.A. Fisk. Scalable parallel application launch on cplant. *Supercomputing, ACM/IEEE 2001 Conference*, 10-16 Nov. 2001.

[4] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. *IDA Center for Computing Sciences*, 1999.

[5] Lei Chai, A. Hartono, and Panda D.K. Designing high performance and scalable mpi intra-node communication support for clusters. Sept. 2006.

[6] T. Hoefler, J. Squyres, W. Rehm, and A. Lumsdaine. A case for non-blocking collective operations. December 2006.

[7] Huang, W. and Santhanaraman, G. and Jin, H.-W. and Gao, Q. and Panda, D.K. Design of high performance mvapich2: Mpi2 over infiniband. *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID 06).*, 2006.

[8] InfiniBand Trade Association. InfiniBand Architecture Specification. http://www.infinibandta.com.

[9] M. Koop, T. Jones, and D. K. Panda. Reducing Connection Memory Requirements of MPI for InfiniBand Clusters: A Message Coalescing Approach. In *7th IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid07)*, Rio de Janeiro, Brazil, May 2007.

[10] M. Koop, T. Jones, and D. K. Panda. MVAPICH-Aptus: Scalable High-Performance Multi-Transport MPI over InfiniBand. In *IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2008)*, April 2008.

[11] M. Koop, S. Sur, Q. Gao, and D. K. Panda. High Performance MPI Design using Unreliable Datagram for Ultra-Scale InfiniBand Clusters. In *21st ACM International Conference on Supercomputing (ICS07)*, Seattle, WA, June 2007.

[12] M. Koop, S. Sur, and D. K. Panda. Zero-Copy Protocol for MPI using InfiniBand Unreliable Datagram. In *IEEE Int'l Conference on Cluster Computing (Cluster 2007)*, September 2007.

[13] Lawrence Berkeley National Laboratory. MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/ index.html, August 2001.

[14] Lawrence Livermore National Laboratory and Hewlett Packard and Bull and Linux NetworX. Simple Linux Utility for Resource Management. https://computing.llnl.gov/linux/slurm/.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[16] A. Moody, J. Fernandez, F. Petrini, and D.K. Panda. Scalable nic-based reduction on large-scale clusters. *Supercomputing, 2003 ACM/IEEE Conference*, 2003.

[17] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, April 1965.

[18] Network-Based Computing Laboratory. MVAPICH: MPI-1 over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/overview/mvapich.

[19] Network-based Computing Laboratory. MVAPICH: MPI over InfiniBband and iWARP. http://mvapich.cse.ohio-state.edu.

[20] Network-based Computing Laboratory. OSU Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks.

[21] OpenFabrics Alliance. OpenFabrics. http://www.openfabrics.org/.

[22] R. Butler and W. Gropp and E. Lusk. Components and interfaces of a process management system for parallel programs. In *Parallel Computing*, 2001.

[23] Sandia National Laboratories. Thunderbird Linux Cluster. http://www.cs.sandia.gov/platforms/ Thunderbird.html.

[24] G. Shipman, T. Woodall, R. Graham, and A. Maccabe. Infiniband Scalability in Open MPI. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[25] A. Shukla and Tim. Brecht. TCP connection management mechanisms for improving internet server performance. *Hot Topics in Web Systems and Technologies, 2006. HOTWEB '06. 1st IEEE Workshop on*, pages 1–12, 13-14 Nov. 2006.

[26] S. Sur, L. Chai, H.-W. Jin, and D. K. Panda. Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[27] Texas Advanced Computing Center. HPC Systems. http://www.tacc.utexas.edu/resources/hpcsystems/.

[28] TOP 500 Project. TOP 500 Supercomputer Sites. http://www.top500.org.

[29] W. Yu and J. Wu and D. K. Panda. Scalable startup of parallel programs over infiniband. In *International Conference on High Performance Computing (HiPC04)*, Bangalore, India, 2004.

[30] W. Yu, Q. Gao, and D. K. Panda. Adaptive Connection Management for Scalable MPI over InfiniBand. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.