Critical Study of Parallel Programming Frameworks for Distributed Applications

A Thesis

presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

by

RUIDONG GU

Michela Becchi, Thesis Supervisor

December 2015
The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

CRITICAL STUDY OF PARALLEL PROGRAMMING FRAMEWORKS FOR DISTRIBUTED APPLICATIONS

presented by Ruidong Gu

candidate for the degree of master of science,

and hereby certify that, in their opinion, it is worthy of acceptance.

__________________________
MICHEL A BECCHI

__________________________
GRANT SCOTT

__________________________
PRASAD CALYAM
DEDICATION

I would like to acknowledge my family, friends and colleagues in Dr. Becchi’s Networking and Parallel Systems (NPS) lab for their understanding, help, encouragement and devotion.
ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my advisor, Dr. Becchi, for her professional guidance and great patience. She helped me define my research goals, giving me a true interest in and dedication to the high performance computing field.

I would also like to thank Dr. Scott and Dr. Calyam for reviewing this manuscript and agreeing to serve on my committee. Accordingly, I also offer my sincere thanks to my colleague, Kittisak Sajjapongse, in the Networking and Parallel Systems (NPS) lab who helped me with my research.
# TABLE OF CONTENTS

Acknowledgments ........................................................................................................... ii

List of Illustrations .......................................................................................................... vii

Abstract .......................................................................................................................... ix

Chapter 1. Introduction ................................................................................................. 1

1.1 Motivation .................................................................................................................. 1

1.2 Primary goals ............................................................................................................ 2

1.3 Thesis methodology ................................................................................................. 3

1.4 Thesis organization ................................................................................................. 4

Chapter 2 Literature Review ......................................................................................... 6

2.1 Introduction to popular parallel programming frameworks ................................... 6

2.2 Related work of programming model comparison .................................................. 11

Chapter 3 Background knowledge on Distributed Applications and Parallel Programming Frameworks ........................................................................................................... 13

3.1 Distributed applications structure .......................................................................... 13

3.2 Parallel programming frameworks overview ......................................................... 14

Chapter 4 IVM Introduction .......................................................................................... 17

4.1 Basic Idea of IVM (Inter-node Virtual Memory) runtime ....................................... 17
4.2 Contribution of IVM layer runtime................................................................. 18
  4.2.1 Load Balancing Scheme ............................................................................ 19
  4.2.2 Memory model.......................................................................................... 22
  4.2.3 GPU Support............................................................................................. 24

Chapter 5 Critical Analysis of Parallel Programming Frameworks ......................26
  5.1 Overview of the main features of the considered parallel programming models ... 26
    5.1.1 Memory model......................................................................................... 26
    5.1.2 Synchronization support and dependency handling ................................. 28
    5.1.3 Dynamic memory allocation support....................................................... 29
    5.1.4 Memory scalability ............................................................................... 30
    5.1.5 Execution model..................................................................................... 32
    5.1.6 GPU support.......................................................................................... 33
    5.1.7 Programming languages versus runtime libraries..................................... 34
  5.2 Programmability and scalability analysis ...................................................... 35
    5.2.1 Programmability ..................................................................................... 35
    5.2.2 Scalability ............................................................................................... 37
    5.2.3 Cluster setup............................................................................................ 39
    5.2.4 Application types .................................................................................. 42
  5.3 Limitations of Current Parallel Programming Frameworks ............................ 43
    5.3.1 GPU support.......................................................................................... 43
5.3.2 Load imbalance on heterogeneous clusters ........................................ 44
5.3.3 Redundant data transfers .................................................................... 45

Chapter 6 Experimental Study of Parallel Programming Frameworks ............ 46
6.1 Experiment setup .................................................................................... 46
  6.1.1 Benchmark applications ................................................................. 47
  6.1.2 Experimental setup ......................................................................... 49
6.2 Performance comparison ......................................................................... 50
  6.2.1 MPI-CUDA and OpenSHMEM-CUDA ............................................... 51
  6.2.2 MPI-CUDA, Charm-CUDA and IVM-CUDA ..................................... 52
6.3 Scalability comparison ............................................................................ 59
  6.3.1 MPI-CUDA scalability and OpenSHMEM-CUDA .............................. 59
  6.3.2 MPI-CUDA, Charm-CUDA and IVM-CUDA scalability ...................... 60
6.4 Solutions and Optimizations of OpenSHMEM ........................................ 61
  6.4.1 Dynamic heap usage ....................................................................... 61
  6.4.2 Limitations in cluster-level execution .............................................. 62
  6.4.3 High-dimension data exchange ......................................................... 62

Chapter 7 Summary and Future work .............................................................. 64
7.1 Guideline of framework choice .............................................................. 64
7.2 Suggestions for future frameworks ......................................................... 66
7.3 Future work ............................................................................................ 69
LIST OF ILLUSTRATIONS

LIST OF FIGURES

Figure 4.1 IVM framework overview .......................................................... 18
Figure 4.2 DS-LB load balancing ................................................................. 19
Figure 4.3 OM-LB load balance ................................................................. 21
Figure 4.4 System design and GPU support .................................................. 24
Figure 6.1 Comparison between MPI-CUDA, OpenSHMEM-CUDA ............... 51
Figure 6.2 Comparison between MPI-CUDA, Charm-CUDA and IVM (DS-LB policy) 53
Figure 6.3 Load distribution across node types (DS-LB policy) ...................... 56
Figure 6.4 Comparison between MPI-CUDA, Charm-CUDA and IVM-CUDA (OM-LB policy) ........................................................................................................ 57
Figure 6.5 Load distribution across node types (OM-LB policy) ...................... 58
Figure 6.6 Comparison between MPI-CUDA, Charm-CUDA and IVM-CUDA .... 60
LIST OF TABLES

Table 3.1 Parallel Programming Framework API (MPI [19] and OpenSHMEM [2]) ..... 16
Table 5.1 Memory model categories ................................................................. 27
Table 5.2 Synchronization categories ............................................................... 28
Table 5.3 Dynamic memory allocation category table ..................................... 30
Table 5.4 Memory scalability categories ............................................................ 31
Table 5.5 Execution model categories ............................................................... 32
Table 5.6 GPU support categories ................................................................. 33
Table 5.7 New languages versus runtime API libraries ..................................... 35
Table 5.8 Programmability comparison table .................................................. 35
Table 5.9 Scalability comparison ....................................................................... 38
Table 5.10 Cluster types comparison table ........................................................ 40
Table 6.1 Hardware setup for MPI-CUDA and OpenSHMEM-CUDA ............... 49
Table 6.2 Hardware setup for MPI-CUDA, Charm-CUDA and IVM-CUDA .......... 50
Table 7.1 Guideline table .................................................................................. 64
Table 7.2 Guidance for design of future parallel program frameworks .......... 67
Parallel programming frameworks such as the Message Passing Interface (MPI), Partitioned Global Address Space (PGAS) languages, Charm++, Legion and High Performance Parallel X (HPX) have been used in several scientific domains – such as bioinformatics, physics, chemistry, and others – to implement distributed applications. These frameworks allow distributing data and computation across the different nodes (or machines) of a high-performance computing cluster. However, these frameworks differ in their programmability, performance, and suitability to different cluster settings. For example, some of these frameworks have been designed to support applications running on homogeneous clusters that include only general purpose CPUs, while others offer support for heterogeneous clusters that include accelerators, such as graphics processing units (GPUs). Hence, it is important for programmers to select the programming framework that is best suited to the characteristics of their application (i.e. its computation and communication patterns) and the hardware setup of the target high-performance computing cluster.

This thesis presents a critical study of established and new parallel programming frameworks, including MPI, PGAS-based languages (OpenSHMEM, Chapel, X10, UPC),
Cham++, Legion, HPX, and Inter-node Virtual Memory (IVM) – a programming system designed and developed at University of Missouri. I first investigate the main features of these programming systems: their memory model, their support for synchronization, their dependencies handling, their mechanisms for dynamic memory allocation, their scalability, their execution model, and their GPU support. I then analyze how these features affect programmability and performance on heterogeneous clusters and for benchmark applications exhibiting different computation and communication patterns. Finally, I develop a benchmark suite where each application is encoded using several programming systems (MPI, Charm++, IVM and OpemSHMEM). The goal of this study is two-fold: first, I want to provide programmers with guidance on the selection of the programming framework which is best suited for their application and cluster setup; second, I aim to provide guidelines for further development of existing and new parallel programming frameworks for distributed systems.
CHAPTER 1. INTRODUCTION

1.1 Motivation

The increasing demand for big data applications requires parallel programming frameworks that allow distributing the computation and data of these applications across multiple compute nodes within high performance computing clusters. Examples of such frameworks include the Message Passing Interface (MPI [1]), Partitions Global Address Space (PGAS) languages (OpenSHMEM [2], Chapel [3], X10 [4], Unified Parallel C (UPC) [5]), High Performance Parallel X (HPX) [6], Legion [7] and Cham++ [8]. Among these parallel computing models, some of them – such as MPI and PGAS - were developed with the goal to provide a unified standard with easy-to-use computation and communication primitives. Other frameworks, like Charm++, were created with specific programmability goals, such as incorporating object-oriented programming principles and providing mechanisms to enable dynamic load balancing. Given its longevity, maturity and portability across operating systems, MPI is currently the most popular distributed programming model, and it has been used in a variety of scientific domains, such as bioinformatics, physics, and computational chemistry. However, MPI is not always an optimal choice. For example, this programming model can lead to poor performance or complex programmability in the presence of load imbalances within the cluster or the application, and in the presence of benchmarks with large data transfers or frequent communication patterns. Hence, it is pivotal for the user to select proper parallel computing models. This thesis provides a comprehensive study of several popular programming models for distributed applications. In particular, our evaluation covers four aspects: scalability, programmability, suitability to different cluster setups, and
suitability to applications exhibiting different computation and communication patterns. First, scalability is important for big data applications deployed on large-scale clusters. Second, the cluster setup (e.g. heterogeneous versus homogeneous systems) can also have a huge impact on performance and programmability. In particular, in this thesis I will focus on heterogeneous clusters that include multicore processors couple with many-core devices, such as Graphics Processing Units (GPUs). Third, different programming frameworks offer programmability-performance trade-offs depending on the computational patterns of the applications (e.g., computation versus communication intensive applications). Last but not least, programmability is also a key consideration.

1.2 Primary goals

The primary goals of this thesis are as follows:

1) Compare representative features of different programming models, such as their memory model, their support for synchronization, their dependencies handling, their mechanisms for dynamic memory allocation, their scalability, their execution model, and their GPU support;

2) Analyze how these features affect scalability and programmability, and what their limitations are under different application and cluster types;

3) Explore features of the PGAS standard, analyze its OpenSHMEM implementation, and discuss optimizations and solutions to some of its limitations;

4) Evaluate Inter-node Virtual Memory (IVM) – a programming system designed and developed at University of Missouri on a variety of applications with different
computational patterns, and leverage the lessons learned from this analysis to contribute to its design;

5) Perform an experimental comparison among MPI, OpenSHMEM, Charm++ and IVM in terms of scalability, programmability, and suitability to different applications and cluster setups;

6) Provide users guidelines on the selection of the parallel programming framework for a specific application type and cluster setup.

7) Provides developers guidelines for further development of existing and new parallel programming frameworks.

1.3 Thesis methodology

As mentioned above, the overarching goal of this thesis is to provide users guidelines on the selection of a parallel programming framework and developer guidelines for further development of existing and new parallel programming frameworks. To this end, this thesis explores the programmability and performance of popular parallel programming frameworks under different cluster and application types.

My critical study considers four aspects: scalability, programmability, suitability to different cluster setups, and suitability to applications exhibiting different computation and communication patterns. In particular, I study central features of parallel programming frameworks that contribute to these aspects; namely: their memory model, their support for synchronization, their dependencies handling, their mechanisms for dynamic memory allocation, their scalability, their execution model, and their GPU support. For example, some of the considered frameworks have a memory model that supports a unified address space. This increases programmability, since it allows users to
manipulate data objects without worrying about their physical location within the disjoint memory spaces among the compute nodes in the cluster.

My analysis can be broken down in two parts: a theoretical and an experimental study. My theoretical study presents a detailed discussion of the main features of the considered parallel programming frameworks. In my experimental study, I implemented a set of benchmark applications using different parallel programming frameworks to evaluate scalability, compare performance, and support my theoretical analysis.

1.4 Thesis organization

Chapter 2 is devoted to a literature survey, which provides background knowledge on popular parallel computing frameworks and presents related research works about programming model comparison.

Chapter 3 provides background knowledge on distributed applications and common parallel computing frameworks.

Chapter 4 covers the IVM framework recently developed in my group at University of Missouri. This chapter presents several examples that show the suitability of IVM to heterogeneous systems. In addition, Chapter 4 also describes how applications benefit from different IVM features, such as memory optimizations, execution model, and abstraction of computing resources.

Chapter 5 presents a critical study of several parallel programming frameworks, focusing on features affecting scalability, programmability, and suitability to different types of applications and cluster setups.
Chapter 6 presents my experimental evaluation of different parallel computing models (MPI, OpenSHMEM, Charm++ and IVM) in terms of scalability, programmability, and suitability to different applications and cluster setups.

Chapter 7 concludes the study with information on how to choose a parallel programming framework according to the four aspects mentioned through this thesis: scalability, programmability, suitability to different cluster setups and application types. Furthermore, based on the result of my analysis and evaluation, I suggest future research directions on parallel programming frameworks.
CHAPTER 2 LITERATURE REVIEW

This chapter reviews popular parallel programming frameworks and some related research works about programming model comparison. The first section mainly covers the development information of selected parallel frameworks today including their developing group, years and highlight features. Then in the second section, several existing related research works of programming model comparison are introduced.

2.1 Introduction to popular parallel programming frameworks

This section provides background information on popular and emerging programming frameworks, namely: the Message Passing Interface (MPI), PGAS languages (Chapel, OpenSHMEM, X10, UPC), HPX, Charm++, Legion and Inter-node Virtual Memory (IVM).

Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. MPI was first proposed in 1991 by a group of researchers; the first version of this standard (MPI-1) was released in 1994. MPI-1 emphasizes message passing and has a static runtime environment. The second version of the MPI standard (MPI-2) includes new features such as parallel I/O, dynamic process management and remote memory operations (which enables one-sided communication). Finally, MPI-3 adds non-blocking versions of collective communication primitives and extensions to one-sided communication primitives. The MPI standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in different computer programming
languages such as Fortran, C, C++ and Java. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications. There are several well-tested and efficient implementations of MPI, including some that are free or in the public. The most well-known MPI implementations are MPICH and Open MPI. MPICH is a freely available, portable implementation of MPI. Open MPI combines technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI), and is used on many TOP500 supercomputers.

**Partitioned Global Address Space (PGAS)** is a parallel programming model that offers a shared address space that simplifies programming while exposing data/thread locality to enhance performance. This facilitates the development of programming frameworks that can deliver both productivity and performance. PGAS has four main features: 1) a local-view programming style, 2) a global address space, where remote references are handled through compiler support, 3) one-sided communication for improved inter-PE performance, 4) support for distributed data structures. Unified Parallel C (UPC), Chapel, X10 and OpenSHMEM implement the PGAS model.

**OpenSHMEM** is an effort to create a specification for a standardized API for parallel programming in the Partitioned Global Address Space. Along with the specification the project is also creating a reference implementation of the API. This implementation attempts to be portable, to allow it to be deployed in multiple environments, and to be a starting point for implementations targeted to particular hardware platforms. It will also serve as a springboard for future development of the API.
OpenSHMEM was created by a group at Cray Inc. in 1993, and was incorporated with SGI’s Message Passing Toolkit, and then was optimized by Quadric Inc. in 1996.

**Chapel** [9] is an emerging parallel language developed at Cray Inc. under the DARPA High Productivity Computing Systems program (HPCS) from 2003 to the present. Chapel supports a multithreaded execution model, permitting the expression of far more general and dynamic styles of computation than the typical single-threaded Single Program, Multiple Data (SPMD) programming models that became dominant during the 1990’s. Chapel was designed such that higher-level abstractions, such as those supporting data parallelism, would be built in terms of lower-level concepts in the language, permitting the user to select amongst differing levels of abstraction and control as required by their algorithm or performance requirements.

**Unified Parallel C (UPC)** is an extension of the C programming language designed for high performance computing on large-scale parallel machines. UPC was developed at University of California in Berkley in 2002. The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor. The goal of the Berkeley UPC compiler group is to develop a portable, high performance implementation of UPC for large-scale multiprocessors, PC clusters, and clusters of shared memory multiprocessors.
X10 is a statically-typed Object-Oriented Programming (OOP) language, extending a sequential core language with places, activities, clocks, (distributed, multi-dimensional) arrays and structure types. All these changes are motivated by the desire to use the new language for high-end, high-performance, high-productivity computing. A team in IBM INC. developed X10 since 2004. The programming model X10 used is called the APGAS model, the Asynchronous, Partitioned Global Address Space model. It extends the standard model with two core concepts: places and asynchrony. The collection of cells making up memory are thought of as partitioned into chunks called places, each with one or more simultaneously operating threads of control.

Charm++ is a parallel Object-Oriented Programming (OOP) language based on C++ and developed in the Parallel Programming Laboratory at the University of Illinois in 1984. Charm++ is designed with the goal of enhancing programmer productivity by providing a high-level abstraction of a parallel program while at the same time delivering good performance on a wide variety of underlying hardware platforms. Processor virtualization is one of their core techniques: the programmer divides the computation into a large number of entities, which are mapped to the available processors by an intelligent runtime system. This separation of concerns between programmers and the system is key to attaining both our goals together. Task migration is also their pivotal feature to achieve automatic load balancing scheme on heterogeneous clusters.

High Performance Parallel X (HPX) is a general purpose C++ runtime system for parallel and distributed applications of any scale. It strives to provide a unified programming model that transparently utilizes the available resources to achieve unprecedented levels of scalability. This library strictly adheres to the C++11 standard
and leverages the Boost C++ Libraries, which makes HPX easy to use, highly optimized, and very portable. HPX is the first open-source implementation of the Parallel X execution model, developed at Louisiana State University in 2011. HPX focuses on overcoming the four main barriers to scalability: 1) starvation- insufficient concurrent work available to maintain high utilization of all resources, 2) latencies- the time-distance delay intrinsic to accessing remote resources and services, 3) overhead- the work required for the management of parallel actions and resources on the critical execution path which is not necessary in a sequential variant, 4) waiting for contention resolution- the delay due to the lack of availability of oversubscribed shared resources.

Legion is a data-centric programming model for writing high-performance applications for distributed heterogeneous architectures. Legion was developed at Stanford University in 2012. Legion has three advantages: 1) user-specification of data properties: Legion provides abstractions for programmers to explicitly declare properties of program data including organization, partitioning, privileges, and coherence. 2) Automated mechanisms: current programming models require developers to explicitly specify parallelism and issue data movement operations. Legion can implicitly extract parallelism and issue the necessary data movement operations in accordance with the application-specified data properties, thereby removing a significant burden from the programmer. 3) user-controlled mapping: by providing abstractions for representing both tasks and data, Legion makes it easy to describe how to map applications onto different architectures. Legion provides a mapping interface that gives programmers direct control over all the details of how an application is mapped and executed. This simplifies
program performance tuning and enables easy porting of applications to new architectures.

**Inter-node Virtual Memory (IVM)** is a programming framework developed at University of Missouri. Its goal is to provide a task-based runtime library that improves programmability of CPU-GPU clusters. The IVM design aims for a balance between abstraction and flexibility. IVM also offers GPU virtualization, which allows homogeneous access to CPUs and GPUs within a high performance computing cluster. Further, it includes an easy-to-use dynamic task creation mechanism to easily embed load balancing in the application. IVM can also be considered a substrate on top of which it is possible to build programming frameworks offering a higher degree of abstraction.

### 2.2 Related work of programming model comparison

There are several related research works that explore programming model comparison from different perspectives.

A group of researchers from University of Washington mainly focus on a comparison between shared memory model and non-shared memory model of programming frameworks [10]. They discuss which memory model fits for the shared memory machine. A comparison of some task-based parallel programming models is discussed by a group at Swedish Institute of Technology. Their research [11] mainly target at task-based programming models including Cilk++ [12], OpenMP [13] and Wool [14] to evaluate their cost for task-creation and stealing, and then to compare their performance using some microbenchmarks. However, these two works focus on programming model itself without thinking the influences from different cluster setups.
and application types. In addition, they also do not investigate other features of a programming framework such as GPU support, load-balancing problems on heterogeneous clusters. Recent research of parallel programming comparison is from an AMD research group, they explore parallel programming models for heterogeneous systems [15]. In this paper, they evaluate the current state of two emerging parallel programming models: C++ AMP [16] and OpenACC [17]. These emerging programming paradigms require minimal code changes and rely on compilers to interact with the low-level hardware language, thereby producing performance portable code from an application standpoint. They analyze the performance and productivity of the emerging programming models and compare them with OpenCL [18] using a diverse set of applications on two different architectures, a CPU coupled with a discrete GPU and an Accelerated Programming Unit (APU). The highlights of this paper consider both GPU support and heterogeneous cluster system. However, they focus on the performance and do not include enough frameworks.
CHAPTER 3 BACKGROUND KNOWLEDGE ON DISTRIBUTED APPLICATIONS AND PARALLEL PROGRAMMING FRAMEWORKS

This chapter provides background knowledge on distributed applications and parallel programming frameworks. Distributed applications split the computation workload across the nodes of a cluster to accelerate the execution. Thus, parallel programming frameworks need to provide mechanisms to split the computation into parts and distribute it to different compute resources, and mechanisms to support data communication among compute resources. We explore similarities and differences among popular programming frameworks.

3.1 Distributed applications structure

In general, distributed applications are broken down into multiple process elements (PEs). Each PE performs its own computation independently on a portion of the data. After the computation is done by all the PEs, a main PE gathers the results. Communication among the PEs is required to exchange data and update common data structures. In addition, synchronization is required to handle data dependencies and access to shared data structures.

Hence, each parallel programming framework must support four basic functionalities: data distribution, computation, communication and synchronization.
3.2 Parallel programming frameworks overview

Current parallel programming frameworks usually distribute workloads among process units, which can be either threads or operating system (OS) processes. The frameworks introduced in this thesis use processes to handle workload. In particular, all these frameworks simplify process spawning and support for shared data.

Existing parallel programming frameworks provide different communication primitives allowing both point-to-point and collective communication (such as one-to-many communication and broadcasting). There are two main communication models: one-sided and two-sided communication. Two-sided communication, typically done through send and receive primitives, requires participation of all communication parties, wherein one-sided communication, which typically is done through the put or get primitives, allows one PE to manipulate the remote data directly without participation of the remote PE. MPI supports both communication models (even if most MPI applications rely on one-sided communication), while PGAS languages (OpenSHMEM [2], Chapel [3], X10 [4]), Legion [4], Charm++ [8], HPX[6] and IVM support two-sided communication.

Conventional parallel programming frameworks typically support barrier synchronization. Barrier synchronization facilitates handling data dependencies within applications. Barrier synchronization may come in two flavors: global and local synchronization. Global synchronization affects all PEs, while local synchronization is limited to a potentially small set of PEs that might reside on a single node. In general, global synchronization mechanisms suffer from larger overhead than local ones. MPI, PGAS languages (OpenSHMEM, Chapel, X10, UPC) and IVM provide global and local
explicit synchronization mechanisms; while Legion, HPX and Charm++ incorporate in their execution model implicit barrier synchronization mechanisms. For example, in Charm++ different code sections execute in a serial manner when the programmer specifies data dependencies across them. In addition, some frameworks, such as MPI, associate implicit synchronization to communication primitives. For example, the $\textit{MPI\_Send}$ and $\textit{MPI\_Recv}$ primitives synchronize sender and receiver automatically. Similarly, collective communication primitives such as $\textit{MPI\_Broadcast}$, $\textit{MPI\_Gather}$, $\textit{MPI\_Scatter}$ implicitly synchronize all the involved parties.
Table 3.1 Parallel Programming Framework API (MPI [19] and OpenSHMEM [2])

<table>
<thead>
<tr>
<th>API call type</th>
<th>OpenSHMEM</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>void *start_pes(0)</td>
<td>void *MPI_Init(int argc, char **argv)</td>
</tr>
<tr>
<td>Process ID</td>
<td>int _my_pe (void)</td>
<td>int *MPI_Comm_rank(MPI_Comm comm, int *rank)</td>
</tr>
<tr>
<td>Communication</td>
<td>void *shmem_double_put(double *target, const double *source, size_t len, int pe);</td>
<td>int *MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);</td>
</tr>
<tr>
<td></td>
<td>void *shmem_double_get(double *target, const double *source, size_t nelems, int pe);</td>
<td>int *MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);</td>
</tr>
<tr>
<td></td>
<td>void *shmem_broadcast32(void *target, const void *source, size_t nlong, int PE_root, int PE_start, int logPE_stride, int PE_size, long *pSync);</td>
<td>int *MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);</td>
</tr>
<tr>
<td></td>
<td>int *MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)</td>
<td></td>
</tr>
<tr>
<td>Synchronization</td>
<td>void *shmem_barrier_all(void);</td>
<td>int *MPI_Barrier(MPI_Comm comm)</td>
</tr>
</tbody>
</table>

Table 3.1 lists a set of primitives belonging to the MPI and OpenSHMEM programming frameworks. These primitives are grouped according to their functionality, as shown in the first column. As can be seen, both programming models include initialization primitives that spawn the required PEs, primitives to obtain the identifier of each PE, point-to-point and collective communication primitives to enable data exchange across PEs, and barrier synchronization to facilitate the handling of data dependencies.
CHAPTER 4 IVM INTRODUCTION

This chapter describes the architecture and contributions of the Inter-node Virtual Memory (IVM) framework.

4.1 Basic Idea of IVM (Inter-node Virtual Memory) runtime

Inter-node Virtual Memory (IVM) is a programming framework developed by my group at University of Missouri. Our goal was to provide a task-based runtime library that improves programmability of CPU-GPU clusters. Our IVM design aims for a balance between abstraction and flexibility. IVM also offers GPU virtualization, which allows homogeneous access to CPU-GPU. Further, it includes an easy-to-use dynamic task creation mechanism to easily embed load balancing in the application. IVM can also be considered a substrate on top of which it is possible to build programming frameworks offering a higher degree of abstraction.

The main goal of the IVM framework project was to increase the programmability of applications running on distributed memory systems that include CPUs and GPUs. This was done by providing a uniform view of memory spaces and compute resources. Specifically, IVM adopts a shared memory model with relaxed memory consistency; the programmer sees a single virtual memory space across all compute resources—CPUs and GPUs. Since all compute resources have access to data objects allocated in this virtual memory space, the programmer can access CPUs and GPUs in a unified way; in the IVM programming interface, all computation resources are represented by device instances.
Under the IVM framework, an application consists of multiple processing elements (PEs). In essence, a PE is a task that is bound to a specific compute resource. Figure 4.1 shows the physical view of a compute cluster consisting of heterogeneous nodes and physical memory spaces (at the bottom) and an abstract view (at the top). IVM provides a view of the virtual memory space containing shared data objects accessed by PEs.

Similar to MPI, a distributed IVM application is started by executing its binaries. Upon instantiation the IVM framework will create a single root-PE, which is the ancestor of all PEs belonging to the application. The root-PE can create its immediate child-PEs and distribute the work to them. It is possible for the child-PEs to further spawn additional PEs. In general, the framework allows a PE to spawn one or more PEs, which can bind to the same or different devices. The IVM framework also provides a mechanism to synchronize PEs. This mechanism consists of wait and signal primitives and is similar to POSIX-threads’ conditional variables. For example, a PE can wait for its children by invoking the wait primitive and specifying the set of PEs to wait on; the child-PEs can then invoke the signal primitive to notify the waiting PE to continue. Programmers can also implement their own synchronization mechanisms using shared memory space in IVM framework.

4.2 Contribution of IVM layer runtime

There are three main IVM contributions including a load balancing scheme, memory model, and one-sided communication GPU support.
4.2.1 Load Balancing Scheme

IVM model acts as a task-based (TB) execution model that supports dynamic process spawning during execution. Hence, IVM model provides load balance through a dynamic process creation (DTC) as follows: Specifically, computing resources with higher computation capabilities are able to spawn new processes when complete current execution is available to handle more work. This mechanism balances workloads and eases their complexity, which allows users to identify faster computing resources according to different applications. In particular, there are two load-balancing schemes based on the DTC mechanism.

**Dynamic Spawning Load Balancing (DS-LB)** – The DS-LB scheme enables work to be dynamically assigned to resources. Figure 4.2 presents a graphical representation of DS-LB. At a high level, this mechanism works as follows. The overall computation is broken into “work-portions,” each executed by a PE. Each device will run a single PE at a time. When a PE finishes executing the work portion assigned to it, it spawns another PE on the same device. Because PEs associated to faster devices will spawn more PEs, more powerful devices will be assigned more work.
Listing 1(a) shows the pseudo-code for DS-LB. IVM API calls are bolded. Lines 6–13 are executed by the root-PE, while lines 16–26 are executed by the child-PEs (either spawned by the root-PE or by other child-PEs). Variable `sync` allows synchronization between root-and-child PEs. This variable, initialized by the root PE (lines 6 and 10), contains a flag for each child PE. The computation is broken in iterations (line 9), and up
to MAX_PEs, PEs are spawned in each iteration (set at line 11). In each iteration, the root PE first spawns a child PE on each device (line 12) and waits for all PEs to complete (line 13). The child PEs use their identification number to retrieve the work portions that they must execute, and perform the computation (lines 18–19). Upon completion, if more work is still pending, each child PE spawns a new PE on the same device where it resides (lines 20–23). Finally, every child PE synchronizes its result with the master copy (line 24), notifies the root PE through the sync variable (lines 25–26), and terminates (line 28).

**Online Monitoring Load Balancing (OM-LB)**—At the high level, the OM-LB scheme performs load balancing by monitoring the performance of different resources and dynamically assigns more work to more powerful devices. Figure 4.3 shows the graphical representation of the OM-LB scheme. The root PE first distributes the work portions equally to all resources as indicated by the solid arrows. During execution, the root PE observes the performances of all PEs and spawns more PEs on the resources that can handle more work portions, as indicated by the pitted arrows. Once the PEs are created, the work-portions will be dynamically redistributed to allow the new PEs to participate in the computation. Differently from DS-LB, in this case all spawned PEs remain active for the entire computation, and each device can be time-shared by the executing PEs.

Listing 1(b) shows the pseudo-code of OM-LB. Lines 6-16 are executed by the root-PE, while lines 18-28 are executed by the child-PEs. The root-PE first allocates memory for variables *time* and *done* along with required data objects (lines 6–8).
Variable *time* is an array used for synchronization purposes and holds the computation times reported back by the child-PEs. The variable *done* is used to notify the child-PEs when the entire computation is completed. The root PE then creates devices, spawns PEs on all devices, and waits for all PEs to report the time (lines 9–10 and 12). Once all PEs have reported time, it determines the performance differences of all PEs and spawns more PEs on the resources, which cause performance differences larger than a threshold (lines 13–16). The child PEs map to data objects, including variables *time* and *done*, and perform work division (lines 18–20). Child PEs that have already worked in previous iterations determine whether the number of total PEs is changed and redivide the work portions as necessary (line 22–23). Finally, all the PEs write back the results and report back the time (line 25–27) The PEs will terminate when the root PE indicates that the entire computation is completed through the *done* variable (lines 16 and 28).

### 4.2.2 Memory model

Figure 4.1 shows shared data objects accessed by PEs through the virtual memory space. A data object can be allocated by one and only one PE. Other PEs can gain access to the data object through a mapping operation. PEs can allocate objects at different times and selectively share the objects with a group of PEs. This property enables applications to define their data sharing schemes and allows the applications to optimize the cluster-wide memory usage and data locality. Upon allocation, each data object is associated with an identifier that can be used by the IVM framework to reference that object. Different PEs can access a data object by providing its identifier to the IVM runtime through the programming interface. Note that data objects allocated in the virtual space do not belong
to any specific PEs. All PEs will have the illusion that the data objects were floating singletons residing in the virtual space.

On the top part of Figure 4.1, the virtual memory space contains two data objects which can be accessed by the PEs. However, the framework needs to manage the physical data corresponding to each virtual data object. The IVM framework manages the data objects by transparently allocating memory for them on different physical spaces. As can be seen in the bottom part of Figure 4.1, there may be multiple physical copies of each (virtual) data object, and those may reside on different devices. If a data object is accessed by multiple GPUs, each of these GPU will require a copy of the data object in its physical memory. Physical copies of data objects can be of two kinds: master-copy and mirror-copy. A master-copy is the copy of the data object that is initially allocated by the instantiating PE, whereas a mirror-copy is a copy that is created upon a mapping operation. Mirror-copies are created only if the allocating PE and the mapping PE are bound to different devices or physical memory spaces. If all PEs are associated to the same device, only the master-copy will be allocated. These physical copies are not exposed to the programmer, who sees only the data objects residing in the virtual space.

The IVM framework synchronizes the master- and mirror-copies of data objects through the put and get primitives. When a PE invokes a put/get operation on a data object, the entire or part of the master- and mirror-copies of the data object are synchronized. The put/get primitives will have no effect for PEs that share the same physical copy. In addition to simple put/get primitives, gather and scatter operations allow for a master-copy to synchronize with different mirror-copies and for a master-copy to distribute contents to different mirror-copies, respectively.
On a node, the framework allocates master- and mirror-copies within Linux’s shared memory. Therefore, data consistency among PEs sharing the same copy is strict. This eliminates the need for duplicating data objects for PEs working on the same device or physical memory space and may reduce the memory footprint of the application.

4.2.3 GPU Support

In the IVM framework, PEs are allowed to share physical data objects (master-/mirror-copies). However, commonly used GPU software stacks (e.g., CUDA [9] and OpenCL runtimes [20]) offer limited support for sharing GPU memory across tasks. In particular, the CUDA runtime associates a different memory address space to each task that uses a GPU. Therefore, a naïve design would allocate on GPU multiple data copies of each shared data objects—one per PE. This solution would be inefficient in terms of memory usage, especially when the data objects are read-only. In addition, it would lead to multiple and unnecessary GPU memory allocations and CPU-GPU data transfers. To address this issue, we include in our IVM framework one additional component: the IVM GPU-daemon. This daemon, shown in Figure 4.4, follows the design of the GPU virtualization runtime system proposed in our previous work [21]. Specifically, the IVM GPU-daemon consists of a front-end library and a backend daemon. The frontend library intercepts CUDA calls and redirects them to the back-end daemon, which decides which

Figure 4.4 System design and GPU support

In the IVM framework, PEs are allowed to share physical data objects (master-/mirror-copies). However, commonly used GPU software stacks (e.g., CUDA [9] and OpenCL runtimes [20]) offer limited support for sharing GPU memory across tasks. In particular, the CUDA runtime associates a different memory address space to each task that uses a GPU. Therefore, a naïve design would allocate on GPU multiple data copies of each shared data objects—one per PE. This solution would be inefficient in terms of memory usage, especially when the data objects are read-only. In addition, it would lead to multiple and unnecessary GPU memory allocations and CPU-GPU data transfers. To address this issue, we include in our IVM framework one additional component: the IVM GPU-daemon. This daemon, shown in Figure 4.4, follows the design of the GPU virtualization runtime system proposed in our previous work [21]. Specifically, the IVM GPU-daemon consists of a front-end library and a backend daemon. The frontend library intercepts CUDA calls and redirects them to the back-end daemon, which decides which
requests should be issued to the CUDA runtime. In the case of IVM, CUDA calls are generated only by the IVM daemon. In fact, applications have a uniform interface to CPUs and GPUs, and access their memories only through IVM API primitives.

As shown in Figure 4.4, the IVM runtime includes two memory paths: one for PEs associated to CPUs, and the other for PEs associated with GPUs. The PEs executing on CPU perform memory allocation and mapping through the sole IVM daemon. The memory-related operations originating from PEs executing on the GPU go first through the IVM daemon and then through the IVM GPU daemon. When a PE performs a memory operation, the IVM daemon determines the compute resource used by the PE. If the resource is a GPU, then the IVM daemon generates the required CUDA calls to complete the operations. These CUDA calls are intercepted by the frontend library and redirect to the backend daemon of the GPU daemon. By controlling all memory operations issued to GPUs, the GPU daemon can avoid multiple physical copies of data objects shared by PEs mapped to the same GPU. This can be done by keeping track of the identifiers of the data objects allocated on each GPU, and selectively ignoring any subsequent \textit{cudaMalloc()} associated with the same data object. In summary, this design bypasses the restrictions of the CUDA runtime and allows multiple PEs using the same GPUs to truly share data objects.
This chapter presents a critical study of MPI, OpenSHMEM [2], Chapel [3], X10 [4], UPC [5], Cham++ [8], Legion [7], HPX [6], and IVM. Specifically, in Section 3.1 we study the main features of these parallel programming frameworks, namely: their memory model, their support for synchronization, their dependencies handling, their mechanisms for dynamic memory allocation, their scalability, their execution model, and their GPU support. In Section 5.2, we analyze how these features affect scalability, programmability and suitability to different application types and cluster setups.

5.1 Overview of the main features of the considered parallel programming models

5.1.1 Memory model

A cluster system consists of multiple nodes (machines). Disjoint physical address spaces exist among both machines and processing elements (PEs). The memory model is the view of memory provided by the programming framework to the applications. The memory model affects the way the programmer performs workload partitioning and describes the communication patterns within the application. In addition, Graphics Processing Units (GPUs) have increasingly become part of high performance computing clusters: incorporating the hierarchical memory model of GPUs into the CPU memory model poses an additional challenge to current programming frameworks.
In Table 5.1, I classify parallel programming frameworks into two categories according to their memory model.

**Table 5.1 Memory model categories**

<table>
<thead>
<tr>
<th>Memory Model</th>
<th>Unified Virtual Memory Space (Superset of DSM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legion, HPX</td>
<td></td>
</tr>
<tr>
<td>IVM, OpenSHMEM, Chapel, UPC, X10, Charm++</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Communication between Objects</td>
</tr>
</tbody>
</table>

PGAS languages support a *unified memory address space*, which enables direct access to data objects for all PEs across the cluster nodes no matter where the PEs reside and the data objects are reside. This technique gives users the illusion that data objects reside in a unified contiguous memory address space without worrying about the disjoint physical memory address spaces of the compute nodes in the cluster system. Most PGAS frameworks handle the data transfers required to support the unified memory address space implicitly. OpenSHMEM, Chapel, X10 and UPC are implementations of the PGAS model and support a unified virtual memory address space. In addition, HPX, Legion, Charm++ and IVM provide unified memory address space as well.

MPI does not support the unified memory address space abstraction. In MPI, each PE operates on a local address space that is not shared with other PEs. As a consequence, initially the *main* PE must distribute the data objects to all the PEs explicitly. Each PE will then handle part of the data objects initially located on the *main* PE.
In hybrid clusters, CPUs and GPUs on the same nodes have different physical address spaces. We will talk about this topic in Section 5.1.6 on GPU support.

5.1.2 Synchronization support and dependency handling

Data dependencies are an important issue in distributed applications, since they may limit the amount of parallelism within an application. All parallel programming frameworks provides explicit or implicit support for barrier synchronization.

<table>
<thead>
<tr>
<th>Table 5.2 Synchronization categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronization &amp; Dependency</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

MPI and PGAS frameworks (UPC, OpenSHMEM, Chapel and X10) allow explicit synchronization through barrier primitives and synchronization variables. Specifically, these synchronization mechanisms can be used to control data dependencies, avoid race conditions among the PEs, and allow accessing shared data in mutual exclusion. Among these frameworks, MPI, UPC and OpenSHMEM offer explicit barrier synchronization primitives. Chapel and X10 use synchronization variables to provide a more flexible synchronization mechanism. Programmers can use synchronization variables to implement different synchronization mechanisms.
HPX, Legion and Charm++ provide implicit synchronization. In particular, HPX aims to eliminate unnecessary global barriers synchronization to reduce overhead. For that reason, HPX distributes the workload among different PEs (or “localities”), and then makes each PE work on its own partitions independently until all PEs have completed. Legion associates access privileges (e.g. R/W-Exclusive, Read-only, Reduction) to data objects and grants those privileges to specific types of PEs to achieve the synchronization. Charm++ adds implicit synchronization primitives to selected function calls and serializes the execution of code sections that have dependencies.

IVM supports both kinds of synchronization: it provides explicit synchronization primitives in the shared memory space and implicit synchronization though the Dynamic Task Creation (DTC) mechanism, which is introduced in Chapter 4.

5.1.3 Dynamic memory allocation support

Dynamic memory allocation has to do with how programming frameworks manage dynamic memory heaps across a cluster system.

In MPI, for example, data objects can be allocated dynamically by a single PE. These data objects will then belong to the PE that instantiated them and cannot be directly accessed by all other PEs unless this PE distributes the data objects to all others.
Table 5.3 Dynamic memory allocation category table

<table>
<thead>
<tr>
<th>Dynamic Memory Allocation</th>
<th>Legion, HPX, Charm++, MPI, X10, Chapel</th>
<th>Only belonging to local PE</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVM, UPC</td>
<td>IVM, UPC [5] - Each PE has ability to allocate memory space dynamically. Programmers can use this space as shared space that has affinity to the allocating process (local to the process).</td>
<td></td>
</tr>
<tr>
<td>OpenSHMEM</td>
<td>Collective call</td>
<td></td>
</tr>
</tbody>
</table>

Like MPI, Legion, HPX, X10, Chapel and Charm++ provide a basic dynamic memory allocation mechanism: dynamically created data objects only belong to their owner PE. OpenSHMEM performs dynamic memory allocation in a collective manner: dynamically allocated data objects are located in the shared memory space and are visible to all the PEs.

Data objects dynamically allocated by PEs are accessible by all the PEs in UPC and IVM as well. Specifically, these data objects are stored in a shared memory address space that has an affinity to the owner PE. Differently from OpenSHMEM, dynamic memory allocation is not a collective call and PEs can access shared data objects only when necessary. Specifically, data objects created by certain PE are stored on its owner node (machine) only at first, and they will be copied into a new node when PEs use *map* primitive to request them. Hence, this technique avoids unnecessary data transfer.

5.1.4 Memory scalability

Memory scalability refers to the ability of the programming framework to handle applications with large datasets without leading to excessive data replication overhead. For example, in MPI all PEs have their own view of data objects, and they may have their
own pieces of data objects, which can exchange data through point-to-point communication (two-sided communication). OpenSHMEM owns unified memory address space that all PEs has a continuous unified view of data objects to allow single PE manipulate data objects without participation of other PEs

**Table 5.4 Memory scalability categories**

<table>
<thead>
<tr>
<th>Memory Scalability</th>
<th>Legion, Charm++, MPI, OpenSHMEM</th>
<th>Memory spacing mapping is done transparently.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVM, UPC, X10, HPX, Chapel</td>
<td>Provides fine grain mapping of memory space and nodes.</td>
<td></td>
</tr>
</tbody>
</table>

Legion assumes that a head node is capable of holding all the data used by the application; the runtime system then distributes portions of data and tasks to nodes transparently. Charm++ require users to initialize the data in an Object-Oriented Programming (OOP) manner at the beginning.

While HPX and PGAS-based frameworks (UPC, X10 and Chapel) allow data objects to be distributed across physical memories, and require users to specify their distribution strategy. So every PE owns part of the data objects at the beginning. Similarly to PGAS-based frameworks, IVM allows data distribution. However, in IVM none of the PEs have ownership over data objects. While a PE task can allocate data, when they terminate they can leave these data in the shared space thus allowing other tasks to selectively gain access to them.
5.1.5 Execution model

We classify existing programming frameworks for distributed applications into two categories according to their execution model: *Cooperative-Process* (CP) and *Task-Based* (TB) frameworks.

**Table 5.5 Execution model categories**

<table>
<thead>
<tr>
<th>Execution Model</th>
<th>Legion, HPX, Chapel, X10 IVM</th>
<th>Task-based (TB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MPI, UPC, Charm++, OpenSHMEM</td>
<td>Cooperative-Process (CP)</td>
</tr>
</tbody>
</table>

In the CP model, a computation is divided into parallel *processes* that execute for the entire lifetime of the application. Processes work cooperatively and exchange information to collectively perform computation. Popular CP programming frameworks include Message Passing Interface (MPI), Unified Parallel C (UPC) [5], Charm++ [8], and Open Symmetric Hierarchal Memory [2] (OpenSHMEM).

In the TB model, PEs do not necessarily remain active for the entire application lifetime and often consist of small portions of code. TB frameworks usually use over-decomposition to break the computation into multiple tasks [22], and they allow programmers to describe parallel computation at a finer grain. TB programming frameworks include Legion [7], HPX [6], Chapel [3], X10 [4] and our IVM.
5.1.6 GPU support

With General-purpose GPU (GPGPU) development, more and more cluster systems contain GPGPU. All the considered frameworks provide basic GPU support. CUDA-aware MPI [1] simplifies the handling of CPU-GPU data transfers from MPI code.

Table 5.6 GPU support categories

<table>
<thead>
<tr>
<th>GPU Support</th>
<th>MPI</th>
<th>GPU API explicitly called</th>
</tr>
</thead>
<tbody>
<tr>
<td>Charm++, HPX, UPC, OpenSHMEM</td>
<td></td>
<td>Limited Support, optimization on GPU</td>
</tr>
<tr>
<td>IVM</td>
<td></td>
<td>GPU sharing, Unification of CPU &amp; GPU</td>
</tr>
<tr>
<td>Legion</td>
<td></td>
<td>Unification of CPU &amp; GPU</td>
</tr>
<tr>
<td>Chapel, X10</td>
<td></td>
<td>Code transfer to GPU code</td>
</tr>
</tbody>
</table>

Charm++, OpenSHMEM, HPX and UPC provide wrapper functions for users to enqueue GPU work and perform CPU-GPU data transfers implicitly, facilitating the integration of CUDA (Compute Unified Device Architecture) code. Some of these systems offer additional features. OpenSHMEM relies on CUDA UVA (Unified Virtual Addressing) to simplify memory handling. The UVA feature, however, is available only on some GPU devices. Charm++ supports pipelined execution of GPU code. UPC enables the shared heap management of GPU device memory.

Chapel and X10 compilers have been extended with CUDA code generation capabilities [23]. In particular, they both help generate CUDA directly from Chapel and X10 language with their compiler to reduce the complexity of CUDA, especially for users without experience in CUDA programming. However, this compiler based CUDA
code generation limits the flexibility that allows users to optimize the CUDA code further.

While Legion provides some form of GPU abstraction, the other frameworks require awareness of the cluster setup and in some cases specific GPUs. IVM, on the other hand, aims to provide uniform access to CPU and GPU resources. To this end, it provides full GPU abstraction: Aside from hiding the location of GPU in the cluster and in the CPU-GPU data transfers, IVM automatically schedules code on GPU resources, provides full memory utilization, and enables GPU to be shared by applications.

5.1.7 Programming languages versus runtime libraries

Two implementation methods of parallel frameworks include the library-application programming interface (Library-API) extension, which is a totally new language. While MPI, HPX, Legion and SHMEM are available as runtime libraries that can be invoked from C programs, Chapel and X10 are full-blown programming languages that are compiled into C, C++ and Java. Hence, the widespread adoption of Library-API and the C program languages is hampered by the need for the programmer to learn and understand their syntax, semantic and execution model, and by the added debugging complexity. Utilizing extensions of C and C++, Charm++ and UPC makes Chapel and X10 adoption easier. IVM is a C runtime library. In addition, given the widespread adoption of MPI, IVM uses initialization, finalization and thread handling primitives similar to those of the C program extensions, while deviating from it in the data and resource management primitives.
5.2 Programmability and scalability analysis

We now analyze how the features describe above affect programmability, scalability, and suitability to different application types and cluster setups.

5.2.1 Programmability

Programmability is an important aspect for any programming frameworks. The memory models, the GPU support and the type of the programming framework (new language versus runtime API extension) affect programmability.

Table 5.7 New languages versus runtime API libraries

<table>
<thead>
<tr>
<th>Framework Nature</th>
<th>Chapel, X10</th>
<th>New Language</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IVM, MPI, HPX, Legion, Charm++, OpenSHMEM, and UPC</td>
<td>Runtime API Extension</td>
</tr>
</tbody>
</table>

Table 5.8 Programmability comparison table

<table>
<thead>
<tr>
<th>Features</th>
<th>Unified Memory Address Space</th>
<th>GPU Support</th>
<th>Runtime API extension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unified Memory Address Space</td>
<td>Abstraction</td>
<td>Code transfer</td>
</tr>
<tr>
<td></td>
<td>MPI</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>PGAS Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OpenSHMEM</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>UPC</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Chapel</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>X10</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>HPX</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Charm++</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Legion</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>IVM</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
As discussed above, there are two main memory models: unified and non-unified memory address space. MPI does not support unified memory address space and requires PEs to explicitly transfer the data as required. PGAS languages (OpenSHMEM [2], Chapel [3] and X10 [4]), HPX [6] and Legion [7] support unified memory address space, which allows each PE to access all the data objects transparently. Specifically, the unified memory address space abstraction let applications have an illusion that data objects are stored in a continuous address space across cluster nodes. Hence, the unified memory address space technique improves programmability significantly.

As far as GPU support is concerned, GPU abstraction and automatic GPU code generation can both improve programmability. IVM and Legion offer GPU abstraction techniques, which provide users with uniform access to CPU and GPU resources. GPU memory allocations and CPU-GPU data transfers are handled automatically by these frameworks as required at runtime without programmer intervention. On the other hand, HPX, PGAS languages (OpenSHMEM, UPC) and MPI offer only basic GPU support: they provide wrapper functions to CUDA code, but require users to be familiar with CUDA programming. Chapel and X10 provide GPU code generation mechanisms within their compiler. This automatic GPU code generation frees programmer from the need to perform CUDA programming.

As mentioned above, MPI, HPX, Legion, IVM and OpenSHMEM are available as runtime libraries that can be invoked from C programs, whereas Chapel and X10 are full-blown programming languages that are compiled into C, C++ and Java. As a consequence, the widespread adoption of these two systems is hampered by the need for
the programmer to learn and understand their syntax, semantic and execution model, and by the added debugging complexity.

In sum, unified memory address space, GPU abstraction, automatic GPU code generation and the availability of a programming framework as a runtime API extension are all features that boost programmability. Legion and IVM support unified memory address space, GPU abstraction and are runtime API extensions. OpenSHMEM provides unified memory address space and works as a runtime API extension. MPI only benefits from its implementation as a runtime library API. Chapel and X10 provide unified memory address space and GPU code generation mechanisms in their compiler.

5.2.2 Scalability
The scalability of the runtime system associated with a parallel programming framework is another essential aspect. In order to use all computing resources available on a high-performance computing cluster, the application performance should scaled up with the size of the cluster.
### Table 5.9 Scalability comparison

<table>
<thead>
<tr>
<th>Frameworks</th>
<th>Features</th>
<th>Data distribution</th>
<th>Dynamic shared data object</th>
<th>Memory isolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PGAS Model</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenSHMEM</td>
<td></td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>UPC</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Chapel</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X10</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HPX</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Charm++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Legion</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IVM</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

The dynamic memory allocation mechanism and memory scalability of a programming framework have an impact on its scalability.

In general programming manner, data objects in MPI [24], OpenSHMEM [2], Legion [7] and Charm++ [8] are stored on the main PE first and then distributed to all PEs. By requiring data copies from a central location, these systems are not scalable to large data sizes. In addition, the flexibility of these frameworks is limited by the fact that dynamically allocated memory belongs only to the creating PE, preventing other PEs from directly accessing these dynamically allocated data objects.

HPX [6] and PGAS-based languages (X10 [4] and Chapel [3]) allow data objects to be distributed across physical memories of a cluster system. This data distribution model solves the limitation of data size. However, the limitation of flexibility still exists because PEs cannot access memory space dynamically allocated by other PEs.
OpenSHMEM [2] supports collective dynamic memory allocation in a shared space that is visible to all PEs. However, data distribution is still done by the main PE and not in a distributed fashion, leading to higher data distribution overhead and limiting scalability.

IVM and UPC allow data to be distributed across the nodes of the cluster. UPC supports shared dynamic data objects. Furthermore, in IVM, none of the tasks have ownership over data structures. While tasks can allocate data, when they terminate, they can leave these data in the shared space thus allowing other tasks to selectively gain access to them. Hence, IVM not only supports dynamic shared data objects, but also isolates the PE with data objects in a shared space that makes data objects in the shared space accessible for use by all PEs in the same machine, reducing the redundant data transfer automatically. In addition, this dynamic allocation is not a collective call.

To summarize, the distribution of data objects across the cluster and the dynamic allocation of shared data objects can improve flexibility and reduce overhead caused by redundant data transfers. Only IVM has all these three features.

5.2.3 Cluster setup

In this thesis, we focus on heterogeneous clusters wherein the computing capability of resources is different across the whole cluster. Heterogeneity can exist both within a node (e.g., a node can comprise CPUs and GPUs) and across nodes (e.g., different nodes can have compute resources with different capabilities). By leading to load imbalances, heterogeneity can lead to resource-underutilization and limit the performance. Hence, it becomes important for parallel programming frameworks to offer support for load balancing.
Table 5.10 Cluster types comparison table

<table>
<thead>
<tr>
<th>PGAS Model</th>
<th>Features</th>
<th>Cooperative-Process</th>
<th>Task-based</th>
<th>Task Migration</th>
<th>Runtime</th>
<th>Custom LB scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>OpenSHMEM</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPC</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapel</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>X10</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>HPX</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Charm++</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Legion</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>IVM</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Since in the CP model, tasks are created at initialization and have the lifetime of the application, CP frameworks support load balancing either through task migration (Charm++, UPC) or by explicitly transferring a load among tasks using communication primitives (MPI, SHMEM) [25-27]. Task migration in Charm++ is performed transparently by the runtime system based on the performance capabilities of the available computation resources and on online profiling. Because in Charm++, data objects must be migrated along with tasks, task migration incurs a data transfer overhead. MPI-2 defines a mechanism to spawn tasks dynamically for load balancing purposes. The use of this feature, however, requires a non-trivial coding effort: because parent and child tasks belong to different MPI groups, the user is left with the burden of using MPI intra- and inter-communicators for communication. In addition, handling communicators incurs runtime overhead.
The TB model supports dynamic task creation and termination, thereby allowing the easy implementation of scheduling and load balancing schemes either in the runtime system or in the application. In Legion and HPX, load balancing is performed by the runtime system. Specifically, Legion handles load balancing in a centralized fashion within the head node, leading to scalability issues. In HPX the dataset is distributed across computation resources, and tasks are then spawned on computation resources with the required portions of dataset. Hence, users cannot tune the load balance by themselves.

While X10 and Chapel do not directly implement load balancing in the runtime, they do allow primitives and mechanisms to embed it in the application. Specifically, users need to find a properly embedded load balancing scheme and then apply it to the applications by themselves, which increases complexity.

Similarly to X10 and Chapel, IVM provides easy-to-use embed custom load balancing schemes in the application. This is possible through DTC, which allows tasks running on a resource to spawn new tasks on a different or same resource and offload computation to that chosen resource. Additionally, it is possible to embed a scheduling component in the IVM runtime and delegate to it the mapping of the new tasks to resources.

In summary, load balancing can be performed automatically by the runtime system or be embedded in the application by enabling dynamic task creation and migration. Embedding load balancing in the application can lead to better flexibility and higher efficiency, which will be demonstrated through our experiments in Chapter 6. However, it may complicate the programming. IVM provides specific load balancing
schemes that can be embedded in the application. X10 and Chapel allow users to embed load balancing in their application, but do not provide specific load balancing schemes.

5.2.4 Application types

We categorize applications into two types: computation intensive and memory intensive, depending on which factor dominates execution. Specifically, two important factors determine the application type: the computational pattern of the application and the data size. In general, applications with little data communication and with complex computation are computation intensive (e.g., N-body simulations). However, applications whose running time is dominated by data access and communication are considered memory intensive (e.g. Himeno benchmark). Although some applications are computation intensive, they can become memory intensive when the data size increases to a certain degree causing data access and communication to dominate the running time.

Compute-intensive applications do not present obvious differences when implemented using different parallel frameworks; however, memory intensive applications are affected by the support for dynamic memory allocation and the memory scalability of the programming framework. Hence, these two aspects affect both scalability and the performance of memory intensive applications in a similar way.

Frameworks that hold data objects at the head node experience higher data transfer overhead and limited scalability, and the data communication overhead is more severe for memory intensive applications. Frameworks that distribute the data across nodes and offer support for shared data objects can reduce data communication frequency and relieve the performance degradation of memory-intensive benchmarks. For example,
the flexible dynamic memory allocation of IVM allows PEs allocated on the same compute nodes to share data objects, thus avoiding unnecessary data transfers.

In summary, memory intensive applications are more sensitive to differences of memory model in frameworks. Specifically, memory intensive applications can benefit from those frameworks with higher scalability.

5.3 Limitations of current parallel programming frameworks

As previous introduction of popular parallel frameworks, limitations still exist in current parallel frameworks compared to Inter-node Virtual Memory (IVM). First, GPU support is limited. Second, load imbalancing problem on heterogeneous system degrades performance of current frameworks as well. Third, limitations of dynamic memory allocation and memory scalability also affect programmability and performance.

5.3.1 GPU support

Most parallel programming frameworks, such as MPI and OpenSHMEM, do not regard GPGPU as a first computing resource as does CPU; hence it is up to the user to handle the complexity of disjoint memory address space between CPU and GPU. For example, OpenSHMEM and HPX just provide basic wrapper function and optimization, but they still require users to be familiar with CUDA programming. Most implementations of MPI do not have GPU support that leave complexity of CUDA programming to users. However, some implementations of MPI support limited GPU support similar with OpenSHMEM. In particular, X10 and Chapel support compiler based code generation for those who do not have any background about CUDA programming; however, users cannot optimize their code further to get fine-tuned CUDA code because all CUDA codes
are generated by relative compilers that cannot be modified. In addition, Legion also has the same GPU abstraction as IVM for users to avoid CUDA initialization and data transfer that improves programmability. However, current parallel frameworks mentioned herein do not support GPU sharing, which prevents better utilization of underlying GPU (introduced in Chapter 4).

5.3.2 Load imbalance on heterogeneous clusters

When considering heterogeneity of a cluster in relationship to computing resources that own different computing capabilities, how to split workload becomes a major problem. Specifically, slower computing resources delay the whole process. Hence, computing resources with higher computing capability need to handle more workloads to achieve load balance.

Early versions of MPI and OpenSHMEM do not support dynamic process spawning and task migration, so the only way for them to solve a load imbalance problem was through profiling. This required users to profile the computing resources in a cluster system at the beginning whereupon the workload could then be divided into pieces with different sizes according to the profiling information about capability of computing resources. However, profiling is time consuming because users always need to profile again once their applications are added to a new cluster system. It is also hard for users to obtain a fine-grained profiling. Newer versions of MPI support dynamic process spawning to tune load balancing automatically, which is more convenient. However, MPI as a cooperative process execution model, has an ongoing process life-span, requiring a big effort from users to enable this dynamic process spawning.
Our tests found that Charm++ supports task migration to enable load balancing, but the result is a high overhead. In particular, HPX and Legion tune load balancing has lower efficiency through a centralized manner of operation. X10 and Chapel as TB execution models also have similar embedded load balancing schemes with IVM. However, these models do not offer instructions on how to use their embedded scheme in guidelines or manuals.

5.3.3 Redundant data transfers

Current parallel frameworks sometimes can encounter overhead due to the need for repeated data transfer [28, 29]. Given the background knowledge of general distributed applications, data distribution is the first step to let each PE get relative data partition. Take matrix multiplication as an example: Matrix A multiple Matrix B and get result Matrix C. Each PE is responsible for part of the computations that make up the resulting Matrix C; hence, each PE in cluster has to obtain part of Matrix A and all of Matrix B. In this case, all the PEs residing on the same machine will copy the entire Matrix B independently, which there will be multiple copies of Matrix B on the same machine. If the size of Matrix B is huge, the data transfer overhead is large and memory occupancy on every machine is high as well, especially the memory capacity on the GPU, which is quite small compared with CPU.

In summary, there are three limitations for conventional parallel frameworks compared with IVM: 1) limited GPU support that increases complexity and limits performance, 2) load imbalance problem on heterogeneous systems, which results in performance degradation, and 3) high overhead due to repeated data transfer that invokes memory capacity problems and performance degradation.
CHAPTER 6 EXPERIMENTAL STUDY OF PARALLEL PROGRAMMING FRAMEWORKS

The first section of this chapter presents experimental environments including benchmark applications and hardware setups. Performance of selected parallel frameworks including Message Passing Interface (MPI), OpenSHMEM, Charm++ and Inter-node Virtual Memory (IVM) are compared in the second section. The third section evaluates scalability of these selected frameworks. In particular, all experiments are based on previous theory analysis. The final section presents solutions and optimizations to some limitations of the OpenSHMEM framework.

Performance is the central metric used to evaluate a framework since performance improvement is the raw power of applying distributed applications to parallel programming frameworks. Scalability is another pivotal metric for parallel programming frameworks. With higher parallel degrees, scalability of a framework reflects how much applications can be scaled up due to higher parallel degrees. However, performance drops when parallel degree of a framework increases to certain point. This is because the overhead of frameworks such as communication overhead and process spawning, increases as well.

6.1 Experiment setup

We pick up four benchmark applications to illustrate our critical study of theory including N-body simulation (NBODY), Dense Matrix Multiplication (DMM), Needleman-Wunsch (NW) and Himeno (HIMENO). The following sections introduce detailed information about benchmark applications and hardware setup.
6.1.1 Benchmark applications

N-body simulation (NBODY)—NBODY is a simulation of a dynamical system of particles. The computation is performed in time-steps. In each time-step, attributes of all particles, i.e., position and velocity, are updated.

IVM implementation of NBODY uses the DS-LB scheme. At each time step, the calculations of particle attributes are divided into a number of work portions, each containing a subset of particles. The child PEs retrieve work-portions based on their identifiers and spawn child PEs dynamically until the overall computation is complete. The MPI-CUDA is very similar to the IVM version except that subsets of particles are statically assigned to tasks; however, MPI-CUDA has no load balancing scheme applied. OpenSHMEM-CUDA also have a similar implementation with the MPI-CUDA version. Charm-CUDA deploys a task migration scheme to achieve load balancing. Specifically, Charm++ incorporates load balancing schemes into runtime itself. Users just need to enable the load balancing scheme inside applications. In particular, Charm++ is a cooperative process (CP), in which a computation is divided into parallel processes that execute for the entire lifetime of the application. Hence, task migration is the proper choice for Charm++ to implement load balancing.

Dense Matrix Multiplication (DMM) — DMM multiplies two square matrices.

Our IVM implementation of DMM uses the DS-LB scheme. We divide the result matrix into $M \times M$ work-portions along both dimensions. The root PE initializes the matrices and spawns the child PEs; the child PEs can then spawn more PEs as necessary. Because DMM involves large data transfers, we use `ivmMapSubset` for the child PEs to map only the required parts of the multiplicand and multiplier matrices. This allows
breaking a single large transfer into multiple smaller transfers, overlapping computations and data transfers, and avoiding broadcasting the whole matrices to all PEs. The tiles that have already been mapped can be reused by other child PEs on the same physical node. Our MPI-CUDA implementation is similar to the IVM version except that tasks are statically assigned portions of the result matrix column-wise and progress down the matrix. OpenSHMEM-CUDA also shares the same implementation with MPI-CUDA. Charm-CUDA uses task migration and deploys similar implementation with MPI-CUDA.

**Needleman-Wunsch (NW)**—NW is a dynamic programming algorithm widely used in bioinformatics for comparing biological sequences [26].

Our IVM implementation of NW uses DS-LB. The root PE and the child PEs perform allocation and mapping of the entire data set, respectively. Sequence-pairs are divided into multiple work portions which can be dynamically retrieved by the child PEs. In the MPI-CUDA version, the sequence-pairs are distributed to tasks equally. OpenSHMEM-CUDA also shares the same implementation with MPI-CUDA. Charm-CUDA uses task migration and deploys a similar implementation with MPI-CUDA.

**Himeno (HIMENO)**—HIMENO is a well-known benchmark application which implements a computational kernel found in the simulation of incompressible fluids. This kernel performs stencil computations on a 3-D grid of pressure values. We divide the computation in work portions along the Z axis of the grid. The MPI-CUDA version assigns to all tasks the same amount of work along the Z axis. OpenSHMEM-CUDA also shares the same implementation with MPI-CUDA. Charm-CUDA uses task migration and deploys a similar implementation with MPI-CUDA.
In the IVM implementation we enable point-to-point communication for exchanging XY-planes by allocating a shared buffer. The sending PEs write into the corresponding slots of the buffer and synchronize the contents with the master copy. Then, the contents in the master copy are distributed to mirror copies to allow the receiving PEs to access the corresponding slots. As the computation proceeds through iterations, the OM-LB scheme keeps performance records of all PEs and spawns more PEs on faster resources.

6.1.2 Experimental setup

We have two separate experiment sets: 1) comparison between MPI-CUDA and OpenSHMEM-CUDA 2) comparison of MPI-CUDA, Charm-CUDA and IVM-CUDA. Accordingly, we have two different hardware setups respectively.

Table 6.1 Hardware setup for MPI-CUDA and OpenSHMEM-CUDA

<table>
<thead>
<tr>
<th># of nodes</th>
<th>Attr</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPU</td>
<td>Intel Xeon® E5620, 2.4 GHz 12 MB cache</td>
</tr>
<tr>
<td></td>
<td>GPU</td>
<td>Nvidia GeForce GTX 480 (Fermi) 480-cores @ 1400 MHz 1.5 GB memory @ 1848 MHz</td>
</tr>
</tbody>
</table>

Comparison experiments between MPI-CUDA and OpenSHMEM-CUDA are on a single-node cluster that includes 4 Nvidia GPUs. Table 6.1 shows the hardware configuration of the compute node.
Table 6.2 Hardware setup for MPI-CUDA, Charm-CUDA and IVM-CUDA

<table>
<thead>
<tr>
<th>Node type</th>
<th># of nodes</th>
<th>Attr</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>2</td>
<td>CPU</td>
<td>20-core Intel Xeon® E5@2.60 GHz, 128 GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GPU</td>
<td>Nvidia Tesla K40m (Kepler)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2,880 cores @ 745 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12 GB memory @ 3000 MHz</td>
</tr>
<tr>
<td>Type 2</td>
<td>2</td>
<td>CPU</td>
<td>20-core Intel Xeon® E5@2.60 GHz, 128 GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GPU</td>
<td>Nvidia Geforce GTX 480 (Fermi)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>480-cores @ 1400 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.5 GB memory @ 1848 MHz</td>
</tr>
<tr>
<td>Type 3</td>
<td>6</td>
<td>CPU</td>
<td>20-core Intel Xeon® E5@2.60 GHz, 128 GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GPU</td>
<td>Nvidia Tesla K20Xm (Kepler)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2,688 cores @ 732 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6GB memory @ 2600 MHz</td>
</tr>
</tbody>
</table>

We conduct our experiments on a ten-node cluster that includes 10 Nvidia GPUs (one per node). Table 6.2 shows the hardware configuration of the compute nodes. As can be seen, the cluster includes three types of nodes that differ in the compute capability of their GPU (i.e., number of cores, memory and core speed). This hardware heterogeneity can cause load imbalance within applications. The 10 nodes are interconnected through a 10 G Ethernet. CentOS 7.1, OpenMPI 1.6.4 and CUDA 7.0 are installed on every node.

### 6.2 Performance comparison

This section has two groups of comparison experiments. The first set of experiments compare performance between MPI-CUDA and OpenSHMEM-CUDA. Performance of MPI-CUDA, Charm-CUDA and IVM-CUDA are compared in the second set of experiments. The popular Message Passing Interface (MPI) is chosen as our baseline.
framework for these experiments. Through critical study based on usage of these frameworks during experiments, we find that the performance of OpenSHMEM-CUDA is not as good as MPI-CUDA; hence, the comparison between MPI-CUDA and OpenSHMEM-CUDA is put into a separate group of experiment in the Section 6.2.1.

6.2.1 MPI-CUDA and OpenSHMEM-CUDA

MPI-CUDA as a baseline does not apply any load balancing scheme and OpenSHMEM-CUDA does not provide any load balancing scheme for users, which they cannot handle load imbalance in a heterogeneous cluster system. Hence, we only use two benchmark applications including HIMENO and NBODY on a homogeneous cluster in Table 6.1.

![Comparison between MPI-CUDA, OpenSHMEM-CUDA](image)

(a) HIMENO application  
(b) NBODY application

**Figure 6.1 Comparison between MPI-CUDA, OpenSHMEM-CUDA**

Performance comparison between MPI-CUDA and OpenSHMEM-CUDA is shown in Figure 6.1. We use various process numbers in both two benchmark applications. Because the cluster system we used in these experiments is small, the data sets are relative smaller compared to the experiments in section 6.2.2. The data sets used
in these experiments have the following sizes: 120 thousand particles for NBODY, and pressure grid of 1027 elements for all directions (X-, Y- and Z-direction) for HIMENO.

For HIMENO in Figure 6.1 (a), MPI-CUDA outperforms OpenSHMEM-CUDA that is about 60% when PE number is 16. The performance improvement gained is about 23% better of MPI-CUDA compared to OpenSHMEM-CUDA when PE number is 8. Through our experimental analysis of running time for each kind of API call in MPI and OpenSHMEM, communication speed advantage of MPI is the main reason that MPI performs better. In particular, HIMENO benchmark has 3-dimensional data communication that increases the gap of communication speed between MPI and OpenSHMEM. Hence, if users port memory-intensive applications with high-dimension data sets to OpenSHMEM that would encounter performance degradation. Hence, MPI-CUDA is selected as our baseline framework.

6.2.2 MPI-CUDA, Charm-CUDA and IVM-CUDA

In our previous critical study of theory, we find that the two main metrics affecting performance of applications in a framework are cluster setups and application types. Given the heterogeneous cluster system used in our experiments, main concern is how the frameworks handle load imbalance on heterogeneous clusters. MPI-CUDA as a baseline framework does not use any load balancing scheme in experiments. Charm-CUDA uses a task migration technique to achieve load balancing. IVM uses both DS-LB and OM-LB schemes to do load balancing according to different benchmark application types.
Figure 6.2 Comparison between MPI-CUDA, Charm-CUDA and IVM (DS-LB policy)

Figure 6.2 shows a performance comparison between IVM, MPI-CUDA, and Charm-CUDA using a varying number of PEs, tasks and chares, respectively. Hereafter, when we can do so without lack of clarity, we will use the abbreviation “PEs” also to refer to MPI tasks or Charm chares. The data sets used in these experiments have the following sizes: 1.2 million particles for NBODY, matrices with 10,000 x 10,000 elements for DMM, and 5,000 sequence pairs for NW. For NBODY and NW, we vary the number of PEs from 10 (the number of nodes and GPUs in the cluster) to 64. For DMM, we vary the number of PEs from 9 to 64 because we divide the work in the resulting matrix equally in both dimensions. In case of MPI-CUDA, the tasks are statically assigned to the GPUs in a round-robin fashion (that is, no load balancing takes place). In case of Charm-CUDA, we enable the refineLB load-balancing scheme provided
by the Charm++ framework. In case of IVM, we use the DS-LB scheme. Specifically, we divided the computation into a number of work-Portions equal to the maximum number of PEs that we wanted to spawn over the execution of the application. Each PE handled a work-portion. Initially the root-PE spawned 10 PEs in the case of NBODY and NW, and nine PEs in the case of DMM; child PEs were then spawned dynamically as needed until the number of PEs was the same as work-Portions. For NBODY and NW, our baseline is the performance of the 10-task configuration under MPI-CUDA. Similarly for DMM, the baseline represents the performance of the 9-task configuration.

Figure 6.2 shows the speedup and slowdown of IVM, MPI-CUDA and Charm-CUDA over the baseline MPI-CUDA implementation as the number of PEs varies. When the number of PEs is equal to that of the GPUs used (10 PEs for NBODY and NW, 9 PEs for DMM), IVM and Charm-CUDA do not perform any load balancing. In this situation, the IVM and Charm-CUDA implementations of all applications achieve the same performance as the baseline. While improving programmability by offering homogeneous access to compute resources and unified virtual memory, the IVM framework shows performance comparable to OpenMPI and Charm++ even in the absence of load balancing.

When the number of PEs exceeds that of GPUs, the GPUs are oversubscribed. MPI-CUDA statically assigns the same amount of work to all GPUs. Charm-CUDA performs load balancing by migrating chares from overloaded to idle resources. IVM performs load balancing by allowing PEs that terminate earlier to spawn more PEs and assign them work. As the number of PEs increases, the size of the work portions decreases, leading to finer-grained load distributions and, thus, higher degrees of load
balancing for both Charm-CUDA and IVM. In all cases where load balancing takes place, IVM performed better than both MPI-CUDA and Charm-CUDA. In addition, the speedup of IVM over MPI-CUDA and Charm-CUDA increased with the number of PEs, and reached an optimal point at 48, 36, and 32 PEs for NBODY, DMM and NW, respectively. The better performance of IVM compared to Charm-CUDA can be explained as follows: First, Charm-CUDA performs load balancing by migrating chares across nodes and GPUs. There are two sources of overhead associated with chare migration: rescheduling of the computation and data transfers between nodes. Through introduction of the IVM memory model in the theory of critical study, IVM minimizes the data transfers by allowing PEs to share data on compute nodes; furthermore, IVM only initiates data transfers when data structure synchronization across PEs resides on different compute nodes as required. We found that, for our applications and datasets, the PEs can share 3–9% of the memory content across the compute nodes, resulting in reduced inter-node data transfers. Second, Charm-CUDA collects profiling information during the first few iterations of each application, and it uses the collected profiling information to perform load balancing in subsequent iterations. Therefore, some iterations in each Charm-CUDA application suffer from load imbalance. IVM, on the other hand, does not require profiling information to perform load balancing. Finally, as discussed below, load balancing in Charm-CUDA may not be optimal.
Figure 6.3 Load distribution across node types (DS-LB policy)

Figure 6.3 shows the relative load distribution across different types of compute nodes for the IVM and Charm-CUDA experiments. Specifically, each bar represents the average load assigned to the nodes of a given type (in percentage terms). For example, in the case of Charm-CUDA with 32 chares, each of the two type1 nodes is assigned about 10% of the overall load. We recall that the presence of heterogeneity among GPUs causes IVM applications using the DS-LB scheme to spawn PEs at different rates, thus introducing load balancing.

Based on characteristics (compute- vs. memory-bound) of the kernels within the considered applications, we expect the performance of different types of nodes to be ordered (high-to-low) as follows: type1-type3-type2 for NBODY and DMM, and type2-type1-type3 for NW. Both IVM and Charm-CUDA can capture the performance...
heterogeneity and are able to distribute a load to compute nodes according to their performance capabilities. However, we observed that in some cases, Charm-CUDA fails to optimally assign the load to the nodes. In particular, in the case of NBODY with 64 PEs, Charm-CUDA assigns its load of type3 nodes equally to type1 and type2 nodes. In addition, in the case of DMM, Charm-CUDA always assigns less load to type1 nodes (equipped with K40 GPUs) than to type3 nodes (equipped with less powerful K20 GPUs). This inefficient load balancing is one of the limiting factors for the performance of Charm-CUDA applications in heterogeneous settings.

![Image](image.png)

**Figure 6.4 Comparison between MPI-CUDA, Charm-CUDA and IVM-CUDA (OM-LB policy)**

The IVM implementation of HIMENO uses the OM-LB scheme. Figure 6.4 shows the speedup of the MPI-CUDA, Charm-CUDA and IVM versions of HIMENO over a baseline MPI-CUDA implementation with 10 tasks (one per GPU). We set the size of the pressure grid to 2,048 elements for all directions (X-, Y- and Z-direction), and we set the threshold described in Section IV.A to 1.5. In the case of MPI-CUDA and Charm-CUDA, we varied the number of PEs (x-axis). Recall that the OM-LB scheme monitors the execution of all PEs and periodically spawns more PEs on faster compute resources and redistributes the work across the PEs mapped to these resources. Therefore, in the case of IVM we are not able to directly control the number of spawned PEs. The numbers on the x-axis represent the maximum number of PEs that can be spawned over execution.
As can be seen, IVM and Charm-CUDA perform similarly to the baseline in case of 10 PEs since in this case, they do not perform load balancing. When the maximum number of PEs is between 32 and 64, IVM outperforms MPI-CUDA and Charm-CUDA and achieves a 2.05 x speedup. In all cases, IVM spawns a total of 25 PEs. We expect this scheme to reduce the communication between IVM daemons for communication-sensitive applications and thereby avoid the DTC overhead.

![Figure 6.5 Load distribution across node types (OM-LB policy)](image)

To understand these results, we monitored the relative load distribution performed by IVM across different types of nodes. Figure 6.5 shows how the load distribution varies in the first few iteration of HIMENO. The numbers above the bars indicate the standard deviation of the execution times of all PEs. We expected this number to be lower once the load became more balanced. In the first iteration the load was uniformly distributed across all types of nodes. In subsequent iterations, the application spawned more PEs on faster type1 and type3 nodes. More powerful GPUs are expected to be time-shared by more PEs. We confirmed this intuition by measuring the standard deviation of the execution time of the PEs. As can be seen, this metric was high during the first few iterations of the application and decreased over its execution, as the OM-LB scheme submitted additional PEs to the more powerful GPUs on type1 and type 3 nodes.
6.3 Scalability comparison

This section evaluates the scalability of selected frameworks including OpenSHMEM, Message Passing Interface (MPI), Charm++ and Inter-node Virtual Memory (IVM). Scalability of OpenSHMEM-CUDA and MPI-CUDA are evaluated independently in the first section. Scalability of MPI-CUDA, Charm-CUDA and IVM-CUDA are evaluated in the second section.

6.3.1 MPI-CUDA scalability and OpenSHMEM-CUDA

We can see the overall running time of HIMENO and NBODY decrease when number of PEs increases from 1 to 8 in both MPI-CUDA and OpenSHMEM-CUDA version in Figure 6.1. Hence, HIMENO-CUDA and MPI-CUDA are both scaled up when parallel degrees increase in a cluster system. However, when number of PEs is 16, performance of HIMENO and NBODY applications drop for both MPI-CUDA and OpenSHMEM version, especially the performance of OpenSHMEM-CUDA for HIMENO decrease around 40%. The overhead caused by large PE numbers (here is 16) comes from two aspects: 1) overhead for process spawning 2) more data communications among the PEs. In particular, overhead of process spawning in these experiments is limited compared to data communication. MPI experiences less performance degradation compared to OpenSHMEM in both NBODY and HIMENO, since the efficiency of handling data communication of MPI is higher. Hence, MPI achieves higher scalability.

In addition, HIMENO as an example of memory-intensive application experiences more fluctuation than NBODY that is a computation-intensive application. For performance gained, MPI-CUDA for HIMENO achieves 60% improvement compared to OpenSHMEM-CUDA version for HIMENO (23% improvement). Similarly,
decreasing percent for HIMENO is also around 40% of OpenSHMEM-CUDA version when PE is 16 that is larger than 5% for NBODY of OpenSHMEM-CUDA version. Hence, memory-intensive applications are more sensitive to communication overhead in parallel programming frameworks than the computation-intensive applications.

**6.3.2 MPI-CUDA, Charm-CUDA and IVM-CUDA scalability**

![Graphs](image)

(a) Nbody application  (b) DMM application  
(c) NW application  (d) Himeno application

**Figure 6.6 Comparison between MPI-CUDA, Charm-CUDA and IVM-CUDA**

The increasing number of PEs leads to a high migration overhead for Charm-CUDA and DTC overhead for IVM, resulting in performance degradation. However, IVM obtained better result and scalability than Charm++, because of the two reasons mentioned in Section 6.2: 1) The advantage of memory isolation in IVM and 2) overhead of task migration in Charm++. This illustrates our critical study in theory about scalability that
the advantage of memory model in IVM leading to higher scalability than Charm++ and MPI.

6.4 Solutions and Optimizations of OpenSHMEM

After conducting experiments using OpenSHMEM benchmark applications, we found three limitations that affect programmability and performance: 1) usage of dynamic heap, 2) problems in cluster-level execution, and 3) high-dimension data exchange in applications.

6.4.1 Dynamic heap usage

Users have to allocate a dynamic heap through a collective call that can easily cause an error. Given the introduction of OpenSHMEM implementations in Table 3.1, there is a dynamic heap for users to define shared data objects through a collective call. However, if a PE does not use this collective call at the same code section when attempting to define data objects, an error will crash the execution. In particular, given the general semantic of a distributed application, data objects need to be initialized by the main PE at the beginning and then distributed into all PEs; hence, allocation for data objects are distinct since each PE might not get all of the data objects. This different allocation mechanism will cause an error in OpenSHMEM. Hence, users need to take care of a collective call when they allocate memory for data objects.

In addition, a collective call for the dynamic heap is not flexible enough within applications. Specifically, data objects are not always shared by all the PEs; however, this collective call requires all PEs to allocate shared space for data objects, which involves inevitable overhead.
6.4.2 Limitations in cluster-level execution

OpenSHMEM framework targets at intra-node cluster systems that only have a single machine. This limitation of the OpenSHMEM framework can lead to a failure when you run applications across a cluster system with multiple machines.

Consider the semantics of an OpenSHMEM framework: Users can share simple data objects that contain a value through global variables to simplify programming, i.e., the size of data objects or an identifier of synchronization. However, this scenario does not work in a cluster system consisting of multiple machines, which can lead to a failure for all PEs not in the main node when attempting to access the correct value of global variables. Specifically, the OpenSHMEM framework does not have an internal mechanism to pass these global variables to all nodes across a cluster system. Instead, runtime just considers the access of local PEs in the main node through the binary code; hence, all PEs residing on other nodes cannot access these global variables correctly.

The solution to this limitation is simple and efficient: Users can define all these global variables in a shared dynamic heap as well through the primitive “shm alloc()” which enable each PE on every node to access the right value initialized by the main PE only.

6.4.3 High-dimension data exchange

Communication among the PEs with high-dimension data objects is time-consuming in OpenSHMEM framework because the nested loops cause higher overhead compared with 1-dimension data transfer. As previously mentioned, users can exchange a 1-D array that is efficient in OpeSHMEM; however, nested loops are required in the presence of a high-dimension array such as 3-D, 4-D or a larger dimension array because there is no fine-
grained feature that enables OpenSHMEM to deal with high-dimension data objects. For example, MPI provides a feature for users to define high-dimension data objects easily, and MPI internally handles this high-dimension data objects exchange in an efficient way.

Overhead of high-dimension data exchange originated from redundant communication calls inside the nested loops. The solution is to remove the nested loops and reduce the number of communication calls. Specifically, the optimization method used here is referred to as a dimension reduction transmission. In particular, the key idea is to convert multiple dimensional data objects into a 1-dimensional one from sender sides and then use single communication call to complete data transfer; accordingly, the receiver recovers the multiple-dimensional data objects. This mechanism reduces the number of communication calls and does not interfere with structures of data objects after data transmission.

Through the experiments conducted in this research, the dimension reduction transmission technique is observed to reduce the overhead and improved performance by almost 50% compared to the nested loop communication pattern. In addition, dimension reduction transmission also involves overhead from dimension conversion steps. Fortunately, dimension conversion overhead is much smaller compared to the high-dimension data exchange overhead caused by nested loop.
CHAPTER 7 SUMMARY AND FUTURE WORK

7.1 Guideline of framework choice

Through critical study of parallel programming frameworks in theory and verifications from experiments, we provide a basic guideline for users to select proper frameworks for their distributed applications. This guideline is based on four main topics. We also have two assumptions for application types and cluster system types: CUDA applications and heterogeneous cluster systems.

Hence, in order to meet the different requirements from users, we are offering Table 7.1 as a guide to help users select their frameworks.

Table 7.1 Guideline table

<table>
<thead>
<tr>
<th>Features</th>
<th>Frameworks</th>
<th>Requirements</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU experience (YES)</td>
<td>MPI, OpenSHMEM, UPC, X10, Chapel, HPX, Legion, Charm++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unified Memory address</td>
<td>OpenSHMEM, UPC, X10, Chapel, HPX, Legion, Charm++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Library extension API</td>
<td>IVM, MPI, HPX, Legion, Charm++, OpenSHMEM, UPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU experience (NO)</td>
<td>X10, Chapel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OOP</td>
<td>X10, UPC, Charm++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU abstraction</td>
<td>Legion, IVM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heterogeneous system</td>
<td>IVM, Charm++</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We consider programmability first because CUDA applications are more complex than CPU versions. Given all frameworks (Message Passing Interface (MPI), OpenSHMEM, X10, Chapel, Unified Parallel C (UPC), High-Performance ParallelX (HPX), Charm++, Legion and Inter-node Virtual Memory (IVM)) analyzed in this thesis, users can select the most applicable framework(s) for them according to the following rules.

For those who are familiar with Chapel or X10 but have no experience about CUDA programming, Chapel and X10 support GPU code transformation through a compiler. Hence, Chapel and X10 are the best choices for making use of GPUs in a cluster system without considering the complexity of writing CUDA codes.

However, Chapel and X10 as new languages compared to other runtime API extension frameworks require more efforts from users. If users have no experience with Chapel and X10 and expect fine-grained CUDA codes as well, they can select from MPI, OpenSHMEM, UPC, HPX, Legion and IVM. In particular, Legion and IVM support GPU abstraction that improves programmability greatly. Hence, for those who are interested in fine-grained CUDA codes and good programmability, they should consider Legion and IVM as their choices.

Object-Oriented Programming (OOP) is also pivotal to some users. Charm++, X10 and UPC all support the OOP model. Hence, those who are interested OOP programming need to choose from Charm++, X10 and UPC.

In heterogeneous cluster systems, only Charm++ and IVM provide specific load balancing schemes for users to choose from. Furthermore, IVM outperforms Charm++
through our experiments, which shows that IVM is a better choice if heterogeneity in a cluster system is obvious. For scalability and application types, features of the frameworks are compatible in that they have similar effects on performance. Through our experiments on scalability, advantage of the IVM memory model makes IVM framework experience a larger scaled up factor compared to Charm++. Hence, IVM also becomes an optimal choice if we consider scalability and memory-intensive application types.

Hence, users can quickly find their appropriate frameworks according to their requirements from Table 7.1, the guideline table. In addition, we can also add new features of frameworks or new frameworks in our guideline table easily to enrich our work.

7.2 Suggestions for future frameworks

From critical study of theory and experiments, we summarize outstanding features of parallel programming frameworks to help future parallel framework design. Seven features can affect programmability, scalability and performance of a framework including the memory model, synchronization, dynamic memory allocation, memory scalability, execution model, and GPU support and framework nature.

With these features in mind, we create a guidance table for future parallel program framework design.
Table 7.2 Guidance for design of future parallel program frameworks

<table>
<thead>
<tr>
<th>Framework attributes</th>
<th>Features</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory model</td>
<td>Unified Memory address space</td>
<td>Programmability</td>
</tr>
<tr>
<td></td>
<td>Memory isolation</td>
<td>Scalability</td>
</tr>
<tr>
<td>Execution model</td>
<td>Task-based model (TB)</td>
<td>Programmability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Suitability to cluster setups</td>
</tr>
<tr>
<td>GPU support</td>
<td>GPU abstraction</td>
<td>Programmability</td>
</tr>
<tr>
<td></td>
<td>GPU sharing</td>
<td>Performance</td>
</tr>
<tr>
<td>Framework nature</td>
<td>Runtime API extension</td>
<td>Programmability</td>
</tr>
<tr>
<td>Advanced features</td>
<td>Distributed data storage</td>
<td>Scalability</td>
</tr>
<tr>
<td></td>
<td>High-dimension data exchange</td>
<td>Programmability Performance</td>
</tr>
<tr>
<td></td>
<td>Object-Oriented Programming</td>
<td>Programmability</td>
</tr>
</tbody>
</table>

The memory model serves as the foundation of parallel programming frameworks and should, therefore, consider both programmability and scalability. First recommendation feature is unified memory address space that can improve programmability. The second recommendation is memory scalability that benefits from memory isolation of IVM. Specifically, data objects allocated by PEs are stored in a shared unified memory address space that does not belong to a specific PE. Hence, PEs on the same machine can share these data objects through the map primitives that reduce unnecessary data transfer. In addition, this map primitives is flexible as well because they do not follow a collective manner, which means only PEs that really need data objects...
require data copy. Hence, unified memory address space and memory isolation are two important features for future framework design from the perspective of memory model.

Execution model is another pivotal feature for parallel programming frameworks. We recommend a task-based (TB) execution model for future framework design. A TB model usually uses over-decomposition to break the computation into multiple tasks, and it allows programmers to describe parallel computation at a finer grain. This mechanism allows dynamic process spawning internally to achieve load balancing instead of task migration with higher overhead. In addition, data distribution is also easier for the TB execution model.

GPU support of parallel frameworks affect both programmability and performance. GPU abstraction is an outstanding feature wherein applications have a uniform interface to CPUs and GPUs, and applications access their memories only through simple IVM primitives. GPU abstraction can accomplish three time saving and security-related tasks as it 1) reduces the complexity for users wanting to initiate GPU, 2) takes care of memory space on the GPUs and 3) takes care of data transfer between CPU and GPU. From the perspective of performance, IVM model also enables GPU sharing where multiple PEs can share GPU resources inside a cluster system. In particular, GPU sharing can fully make use of underlying GPU hardware resources. Hence, GPU abstraction and GPU sharing are two advanced features that free users from the complexity of GPU initialization.

Designers of future parallel frameworks also need to decide their framework nature. Runtime API extension is a good way for users to adapt their standard C or C++ codes to a new framework. However, sometimes runtime API extension may have some
constraints and extra overhead that designers need to develop a new language. In particular, we suggest that designers make sure the new language follows the C standard and is easier for users to learn and use.

Besides these features, it is also important to recommend two excellent features developed in the critical study: These are: 1) data distributed storage that can increase scalability for frameworks to handle large data sets and support some special application types, i.e., graph applications and 2) high-dimension data exchange, which is also a useful feature for those applications with higher dimensional data. Dimension reduction technique can handle high dimensional data exchange with lower overhead (see the section 6.4.3).

Object-Oriented Programming (OOP) model is also required by some C++ standard applications, and framework designers may also take this into consideration. Our main purpose in investigating OOP was to define what programs are compatible with OOP (Charm++, X10 and UPC) and thereby give straightforward guidance to those wanting to know what can be used with OOP when developing or using applications.

### 7.3 Future work

The future work mainly covers two aspects. First part is the further works for parallel programming frameworks. Then extensions for Inter-node Virtual Memory (IVM) model are the second part.

This thesis involves four benchmark applications. In the future, we plan to have at least eight benchmark applications to support our findings. The framework implementations researched herein are Message Passing Interface (MPI), OpenSHMEM,
Charm++ and Inter-node Virtual Memory (IVM). Hence, framework implementations will be expanded in experimental part as well.

IVM model currently can offer good programmability, scalability and suitability to different cluster setups and applications. In the future, we plan to add new functionalities for IVM model. Automatic load balancing will be offered in the future IVM to provide embedding load balancing schemes automatically by framework itself, improving programmability further. Then we also decide to implement a scheduling mechanism inside IVM to support execution of multiple applications concurrently in a cluster system with IVM. This technique can make use of underlying hardware in a cluster system.


