



Improved Address Selection for Message Passing in Multi-Domain Cluster Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Informatiker

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA
Fakultät für Mathematik und Informatik

eingereicht von Thomas Peiselt
geboren am 22.09.1980 in Gera

Betreuer: Dipl-Inf. Christian Kauhaus
Prof. Dr.-Ing. Dietmar Fey

Jena, 30. September 2007

Improved Address Selection for Message Passing in Multi-Domain Cluster Systems

Kurzfassung

Clustercomputer finden insbesondere im Bereich des High-Performance Computing immer weitreichendere Verbreitung, insbesondere aufgrund ihres hervorragenden Preis-Leistungsverhältnisses. Wenn die durch einen Cluster zur Verfügung gestellte Leistung für gewisse Berechnungen nicht ausreicht, eine Erweiterung der vorhandenen Kapazitäten aber aus finanziellen, räumlichen oder zeitlichen Gründen nicht praktikabel erscheint, bietet es sich an, mehrere solcher Systeme zu einem Multi-Domain Cluster zu koppeln. Message Passing Systeme benötigen eine Möglichkeit, Nachrichten zwischen beliebigen, an der Berechnung teilnehmenden Rechnerknoten auszutauschen. Da die einzelnen Clustersysteme oft, aufgrund der Knappheit von öffentlichen IPv4-Adressen, mit privaten Adressen ausgestattet sind, besteht auf Netzwerkebene mit IPv4 oft keine Möglichkeit, diese Ende-zu-Ende Konnektivität herzustellen. Eine mögliche Lösung ist die Verwendung von IPv6, da hier genügend Adressen zur Verfügung stehen. Die entstehenden Netzwerktopologien mit gemischter IPv4/IPv6 Adressierung können eine relativ hohe Komplexität aufweisen, und es ist eine bisher wenig untersuchte Problemstellung, in einem solchen System die richtigen Zieladressen auszuwählen, um eine erfolgreiche Kommunikation in Message Passing Systemen sicherzustellen. Gegenstand dieser Diplomarbeit ist daher die Entwicklung einer Auswahlheuristik, um die Kommunikation in komplexen Multi-Domain Clustersystemen zu ermöglichen, und möglichst automatisch und ohne administrativen Aufwand durch Clusterkopplung dem Anwender eine erhöhte Rechenleistung zur Verfügung stellen zu können.

Abstract

Cluster computers enjoy an increasing popularity, especially in the domain of high-performance computing. The major reason is their excellent price-performance ratio. When the performance provided by a cluster system is no longer sufficient, and an expansion of the existing system is not feasible due to space, economic or time reasons, it is desirable to combine several existing cluster systems into a multi-domain cluster. Message passing systems need the capability to exchange information between all participating compute nodes, but IPv4-based network setups are often not able to provide this capability directly, due to the fact that public IPv4 addresses are scarce and most cluster installations use private, unrouted IPv4 addresses for internal communication. One possible way to establish the desired end-to-end connectivity is the deployment of IPv6 on the cluster systems, because there is no lack of such addresses. However, the resulting network topologies with mixed IPv4/IPv6 addressing might exhibit high complexity. The problem of selecting the best way to connect to a peer process in such a setup has not been solved satisfactory so far. Subject of this diploma thesis thus is the development of an address selection heuristic to enable communication in multi-domain cluster systems, to offer the user increased performance without administrative intervention.

Contents

Abkürzungsverzeichnis	7
1 Introduction	8
1.1 Motivation	8
1.2 Cluster of Clusters	9
2 Open MPI	15
2.1 Project Goals	15
2.2 Structure of Open MPI	16
2.2.1 Open Portable Access Layer	16
2.2.2 Open Run-Time Environment	17
2.2.3 Open Message Passing Interface	18
2.3 Open MPI communication frameworks	19
2.4 Open MPI Startup and Current Address Selection	20
2.4.1 Exchanging Process Connection Information during Startup	20
2.4.2 Current Address Selection Method	20
2.4.3 Problems with the Current Approach	21
3 Improved Address Selection	23

<i>CONTENTS</i>	6
3.1 Design Goals	23
3.2 Connectivity in Private Networks	24
3.3 Solution to the Address Selection Problem	26
3.3.1 Modelling the Network Setup as a Graph	27
3.3.2 Properties of a Desired Solution	29
3.3.3 Solutions for the Assignment Problem	30
3.3.4 Generating All Permutations	31
3.4 Implementation	32
4 Evaluation	35
4.1 The Virtual Test Environment	35
4.2 Assuring Connectivity	36
4.2.1 Single Cluster Systems	36
4.2.2 Multi-Domain Clusters	39
4.3 Striping	41
5 Summary	43
Appendix	47
A CD contents	48

List of Abbreviations

Kürzel	Beschreibung
IP	Internet Protocol
MAC	Media Access Control
MPI	Message Passing Interface
NAT	Network Address Translation
NFS	Network File System
SSH	Secure Shell
TCP	Transmission Control Protocol
VM	Virtuelle Maschine
VPN	Virtual Private Network
VTE	Virtual Test Environment
MDC	Multi Domain Cluster
GPR	General Purpose Registry
BML	Byte Management Layer
PML	Point-to-Point Management Layer
BTL	Byte Transfer Layer
OOB	Out of Band Communication

Chapter 1

Introduction

1.1 Motivation

Traditionally, natural sciences are based on two methods: develop a theory and experimentally verify it. The technological advance in the area of computer hardware leads to a significant rise in the computational power available, eventually offering a third method: The simulation. Historically, the fastest computers were based on specialised hardware and proprietary software, carrying a price tag which made these mainframes unaffordable for most research institutes and smaller industry. Today, cheap common off-the-shelf hardware (COTS) can be used to obtain reasonable computational power by combining a number of personal computers connected via standard ethernet or more advanced networking technology, like Infiniband or Myrinet [3]. Such a system is usually named a Cluster Computer, or Cluster for short. The advantages of a system built from standard hardware are obvious:

- cheaper hardware
- cheap and simple upgrading by adding more compute nodes, so it is possible to start small and expand later
- reduced maintenance and service costs by using hardware and software available everywhere

The trend towards cluster computers is also reflected in the list of the top 500 fastest computers [7], published semiannually. In the year 2000, only

2% of the supercomputers listed in the top 500 were clusters, in 2003 the share grew to 30% and has now stabilized at almost 75% (as of June 2007) [21]. The two most common processor manufacturers found in the top 500 list are Intel (58%) and AMD (21%) - their products can also be found in an average personal computer. Of course, such systems are still not affordable for the average business, but the numbers drawn from the top500-list clearly show that the concept of using standard pc components is a feasible method to build high-performance computers, as the underlying concept can also be applied to smaller systems. In fact, Gbit ethernet is the most used internal system interconnect technology [2].

With the growing size of cluster systems, the involved network technology is becoming increasingly sophisticated. In the realm of high-performance computing, using several interconnects for each compute node to increase the communication performance is becoming more common. This results in more complicated network topologies where multiple communication paths exist to reach peer nodes. To be able to use these additional resources efficiently, cluster nodes need to be aware of the capabilities of the available networks and make the best choice for maximum resource usage and communication performance.

In this work, we improve an existing solution to the challenge of finding the best way to connect to other cluster nodes, to allow cluster systems to take full advantage of the potential of the available interconnects.

1.2 Cluster of Clusters

The increasing demand for computational power in almost all areas of scientific computing has led to a large number of small to medium-sized cluster installations, many of which use standard 1000 Mbit/s ethernet and the TCP/IP protocol for communication. Even in our faculty, there are at least three independent cluster computers, each with 8-32 compute nodes, all operating independently. Due to limited budgets and lack of building space, extending the existing clusters is not always an option, especially when the needs for higher performance are short-term e.g. for a certain project or simulation.

In such a case, bundling several cluster computers to form a cluster of clusters, or multi-domain cluster (MDC) is a desirable option to increase the peak performance, allowing to undertake larger simulations, increase

accuracy of the calculations or simply get your results faster. In contrast to ordinary clusters, a multi-domain cluster has:

- increased performance
- increased utilisation
- heterogenous hardware
- possibly complicated network topology

To run a parallel application on multiple compute nodes, it is usually necessary to send messages from each compute node to some or all other nodes. This includes sending data calculated by one node which is needed by other nodes for further calculations, synchronization between nodes or collecting node information. However, it is quite often impossible to directly reach a compute node within a cluster from a computer outside of its local network, a result of the fact that a typical cluster network setup consists of a private IPv4 network according to RFC1918 [26]. Since the number of public IPv4 addresses is limited, it is often impossible to obtain a public IPv4 address for each compute node. To be able to run an application in a multi-cluster environment, each compute node must be able to reach each other compute node. Several solutions to enable end-to-end connectivity over cluster boundaries have been proposed:

Virtual Private Network (VPN): By integrating all clusters into a virtual private network, the system administrators can achieve full end-to-end connectivity between all participating compute nodes. A VPN-based solution does not require modifications to the underlying MPI or PVM library, because packet routing is handled transparently by the network layer. The major disadvantage of this approach is the necessity to modify the complete network configuration [18] to assign a unique private IPv4 address to each participating node. Adding another cluster at a later stage requires additional reconfiguration on all networks already integrated into the VPN. Generally, VPN-based solutions suffer from a slight performance drop because of the increased protocol overhead resulting from sending IP traffic through a tunnel. Stiffer performance penalties can occur especially when the traffic inside the VPN is encrypted [20].

Proxy-based Solutions: Deploying user space daemons to close the gap between disconnected networks has been frequently used, for example in systems like PACX, ePVM, Stampi or MPICH/Madeleine [15, 25, 29, 9].

Every cluster participating in the multi-domain cluster is interpreted as a cell, where traffic inside a cell is directly sent to the receiving compute nodes. All other messages are directed at the local proxy (which has a public IPv4 address), which forwards it to the proxy responsible for the cell where the receiving compute node resides. The remote proxy is then able to transfer the message to the destination host. Proxy-based solutions require massive modifications in the underlying communication libraries, as the compute nodes network code needs to support a multi-cell setup. Manual configuration is required to adjust to the cell setup, and adding or modifying one cluster demands configuration changes at all sites. Messages sent across cluster boundaries also suffer from a significant latency penalty: the message must be handled by the proxy server in user space, so it must traverse the complete kernel TCP/IP stack, is then assigned a new destination address after determining the correct remote proxy server and is finally handed back to the kernel's TCP/IP stack. The same procedure must be applied at the remote proxy server, so each message faces two additional kernel–user–kernel round trips [18]. The ePVM extension to PVM faces a performance loss of up to 75% [20], which is not acceptable especially for communication-demanding parallel applications.

The configuration overhead and performance loss for both VPN-based and proxy-based solutions is generally not acceptable in the realm of high-performance computing. Direct end-to-end connectivity between all participating compute nodes without the need for special processing of messages crossing cluster boundaries is desirable. This is referred to as the single-cell concept: There is a transport available to reach all hosts. Since public IPv4 addresses are not easily obtainable, especially for larger numbers of nodes, a viable alternative is to use the IPv6 protocol [26]. The IPv6 protocol offers a 128bit address space, which translates to $3.4 \cdot 10^{38}$ addresses or $6.5 \cdot 10^{23}$ addresses per square meter, a number large enough to eliminate address shortage in the future.

Most existing IPv6 implementations of PVM or MPI libraries are IPv6-only and are thus not capable of handling the majority of current cluster installations [27, 23]. In a recent work, the university of Jena extended the Open MPI library [4, 19] to support both IPv4 and IPv6, including mixed environments where both IPv4 and IPv6 are necessary to communicate between certain compute nodes.

The benefits of using IPv6 to connect multi-domain clusters:

- Direct end-to-end connectivity: Once IPv6 is deployed on all sites,

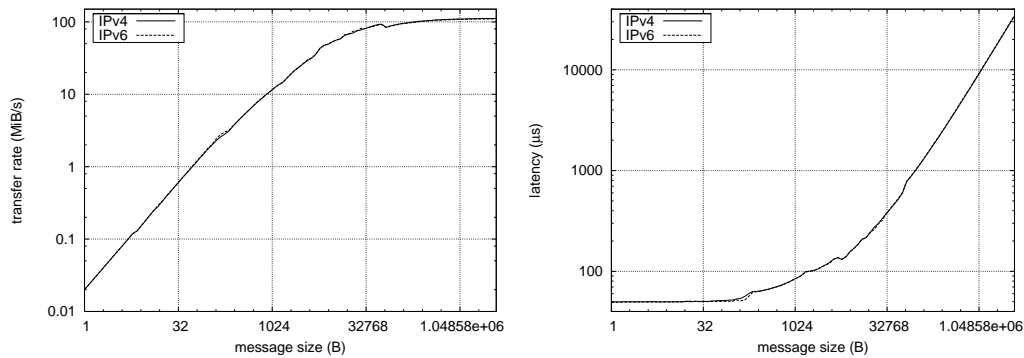


Figure 1.1: Intra-cluster ping-pong benchmarks on Gigabit ethernet using IPv6-enabled Open MPI, taken from [19]

there is no need to configure proxies, set up VPN tunnels or alter your network setup. Every compute node is reachable directly, adding or removing nodes requires no administrative intervention – just add a node to the list of execution hosts and start your application!

- High performance: Since IPv6 routing is handled in the same layer that also handles IPv4 routing, there is no bottleneck when sending messages over cluster boundaries. A slight performance penalty can be expected due to the increased header size for IPv6 (40 B) compared to IPv4 (20 B). In fact, the bandwidth measures for the IPv6-enabled Open MPI on a 1 Gbit/s ethernet link show almost no difference between these two protocols: 110.0 MiB/s throughput for IPv6 and 111.5 MiB/s for IPv4, as shown in [19]. Latency figures depicted in Figure 1.1 showed no measurable difference, making IPv6 a viable alternative even for intra-cluster communication.

To be able to connect a multi-domain cluster via IPv6, the following conditions must be met:

- The message passing library must support IPv6, preferably a mixed IPv4 / IPv6 environment. The next production release of Open MPI [4] will satisfy this condition.
- IPv6 must be deployed on all participating sites and IPv6 traffic must be routed between them. Even though IPv6 deployment is currently not very common on the internet, it should be possible to deploy it

Solution	Modifications to the Message Passing library	Administrative Overhead	Performance
VPN-based	no – the VPN client and network routing transparently handle traffic between clusters. Direct end-to-end connectivity is established.	All sites must synchronize their clusters' address spaces. Reconfiguration on all sites needed whenever a cluster is added	Depends on the specific implementation, kernel-based vpn solutions can have competitive performance
Proxy	yes – due to the multi-cell setup, the message passing library must treat inter-cluster communication differently	Address space synchronisation not needed, but modifications to one of the clusters usually demands administrative intervention on all sites.	Massively increased latency due to overhead from kernel-user space-kernel roundtrips.
IPv6	yes – the message passing library must be able to handle IPv6 peer addresses.	IPv6 must be deployed on all sites and on the routes between them.	Limited by the speed of the connection, comparable to a direct IPv4 connection.

Table 1.1: Comparisation between different concepts to connect a multi-domain cluster without full public IPv4 connectivity

inside a faculty, university or similar administrative domain. The network of our faculty already has IPv6 deployed, so in our scenario it is already possible to combine our three clusters to form a multi-domain cluster. Additionally, many governments have already started local initiatives to boost use of IPv6. The US department of defense recently decided to have IPv6 fully deployed in 2008, and the german DFN started a project named 6win [1] to connect all german universities and research institutes with IPv6. If IPv6 is not yet available, it is still possible to connect sites via tunneling IPv6 traffic inside IPv4. However, this solution is not optimal, as it results in larger protocol overhead, comparable to the overhead resulting from routing traffic through VPN tunnels.

With IPv6 deployed and an IPv6-enabled message passing library available,

multi-domain clusters can be set up with little to no administrative intervention, and without considerable performance loss. Open MPI already supports basic IPv6 functionality [20], but the challenges arising in heterogeneous network environments with complicated topologies are not yet solved satisfactorily: Each compute node must be able to determine the correct and best-performing way to send messages to all peer processes, preferably without manual intervention. In this work, we present an improved address selection algorithm capable of automatically handling mixed IPv4 / IPv6 environments and complicated network topologies.

Chapter 2

Open MPI

In this chapter, we present an overview of the Open MPI project. Section 2.1 contains a short summary of the project goals, Section 2.2 sketches the component-based architecture used for all functional areas of Open MPI, and in Section 2.3 we have a more detailed look at the components and modules dealing with communication requests. Finally, in Section 2.4, we analyse the address selection method currently utilized and discuss the problems that might occur with certain network setups.

2.1 Project Goals

Open MPI is an open source MPI implementation aiming to fully implement the MPI-2 [24] standard. The project started in 2004, aiming to become “the best MPI library available” [14]. Developed by a wide range of partners with academic and industry background, it attempts to combine new, state-of-the-art concepts from the high-performance computing domain with many years of experience from previous MPI implementations, namely FT-MPI [8], LA-MPI [16], LAM/MPI [10] and PACX-MPI. However, Open MPI is not just a merger of existing code bases, but a completely new project started from a blank sheet of paper.

Open MPI is designed to be portable to a large number of different platforms, ranging from small clusters to large installations with thousands of compute nodes. The ability to run in heterogeneous environments is an important functionality to create multi-domain cluster systems. Its component-

based architecture offers flexibility, easy extendibility, run-time configuration and dynamic adaption to the environment.

2.2 Structure of Open MPI

The major underlying concept of Open MPI is the Modular Component Architecture (MCA). Functional areas within Open MPI are represented by MCA *frameworks*, where each framework stands for a single, targeted purpose and has a well-defined public interface. There are frameworks for collective operations, error management, network access and resource management. For a complete list of Open MPI frameworks see [5].

An MCA *component* stand-alone collection of code which implements the interface defined by a framework. It can be plugged in at compile-time or run-time to add or change Open MPI's functionality or behavior. Finally, a module is an instance of a component.

For example, consider the byte transfer layer (BTL) framework, which abstracts access to sending and receiving data over different kinds of networks. There are several components for the BTL framework, such as `btl/openib` to support Infiniband, `btl/tcp` for transferring data over TCP/IP networks, or `btl/sm` which uses shared memory to transfer data to a different process running on the same host. At startup, each component creates modules depending on the identified hardware. If there are several NICs available, the `btl/tcp` component would instantiate several modules, where each module handles one ethernet card. Adding support for a new transport can thus be achieved by creating a new BTL component, which can be developed and maintained independently. It does not even need to be under an open source license, it is sufficient to supply a shared library containing the component at run-time.

Open MPI is divided into three subprojects: The Open Portable Access Layer (OPAL), the Open Run-Time Environment (ORTE), and the Open MPI (OMPI) project.

2.2.1 Open Portable Access Layer

The Open Portable Access Layer mainly provides utility and glue code for both ORTE and OMPI, and offers portable access to the operating system.

It handles object management, datastructures (lists, hash tables, ...), memory management and network interface discovery. It contains an event library that supports invoking callback functions based on certain events, e.g., changes to a socket state or timeouts. To allow for concurrent programming, threading is supported, e.g. for message passing in the background while proceeding with the computation.

2.2.2 Open Run-Time Environment

The Open Run-Time Environment [11] provides services for process launch (including error management when processes fail), resource management (resource discovery and allocation) and interprocess communication. An ORTE system consists of computational resources, which are aggregated into an ORTE universe. An ORTE universe consists of a number of cells, where each cell has a common point of contact or process spawn mechanism [11]. For example, a single cluster system can be considered a cell from the ORTE perspective, as such a system has a job management mechanism capable of spawning and stopping processes on all individual nodes.

ORTE consists of several core functionalities:

General Purpose Registry: A central unit of ORTE is the general purpose registry (GPR), a data store containing data to be shared between all running processes. Each entry in the GPR is a key-value pair, arranged in so-called named segments and further divided into containers, according to their functionality. Users can add segments and containers depending on their requirements. The resulting structure is a searchable index offering access to arbitrary data during run-time. It offers a publish/subscribe mechanism for event-driven applications, allowing processes to specify under what conditions and how they wish to be notified in case of modifications to the GPR.

Resource Management: The resource management system is subdivided into the resource discovery subsystem (RDS), the resource allocation subsystem (RAS), the resource mapping subsystem (RMAP) and the process launch subsystem (PLS) The RDS identifies the computational resources available by reading user-supplied hostfiles and cell information. The RAS examines the command line and the environment to identify resources already allocated, and attempts to obtain an allocation from a suitable cell based on information from the RDS. When the cell does not provide a resource mapper, RMAP is used to map the application's processes to the

allocated resources (nodes). After resources are discovered, allocated and the requested processes are mapped onto the available resources, the PLS is responsible for starting the processes on the compute nodes. It first spawns a head node process (HNP) on the frontend machine of the appropriate cell, which sets up the environment needed for the remaining processes, which are started afterwards.

Error Management: Error management can be differentiated into two tasks: First, monitoring processes and identifying error conditions and second, appropriate handling when an error occurs. The first task is handled by the State-of-Health monitor (SOH), using components specific to the local environment. If an error is detected, the error manager (EMGR) is called to determine the proper response. It can either be called locally (when a local process detects an error) or globally (when a process is terminated unexpectedly).

Support Services: The support services provide basic functionality for the application and the other ORTE subsystems. This includes the name services subsystem (NS) to supply each process within the application with a unique identifier, the Run-time Messaging Layer (RML) for sending and receiving administrative messages (also used by the GPR publish/subscribe mechanism to notify processes), and finally the I/O Forwarding subsystem (IOF) to forward the process I/O to and from the user.

2.2.3 Open Message Passing Interface

The OMPI subproject contains the actual implementation of the MPI-2 standard, and currently supports bindings for C, C++, Fortran77 and Fortran90. It covers all MPI-related functionality, including frameworks for collective operations, topology support, MPI-I/O, one-sided communication and, of course, several frameworks responsible for sending and receiving point-to-point messages.

We summarized how the Open MPI project is structured into subprojects and how the functionality is divided between them. In the following section, we look more closely into the frameworks (and the corresponding component implementations) which handle the communication between processes, as understanding these components is especially important for our work.

2.3 Open MPI communication frameworks

In Open MPI, there are three different MCA frameworks that deal with sending and receiving messages [17]:

1. Point-to-Point Management Layer (PML)
2. Byte Management Layer (BML)
3. Byte Transfer Layer (BTL)

The PML framework is responsible for sending and receiving messages on a high abstraction level. It defines an interface which maps closely to the MPI interface. In fact, the actual MPI point-to-point messaging functions provide no significant functionality and are mere function wrappers for the PML implementation with optional argument validation [30]. The PML handles message fragmentation and schedules messages across the available transports, which are abstracted and represented by the other frameworks. Of course, it also takes care of reassembling received messages. During the execution of an application with Open MPI, there is always exactly one PML module active, which handles all MPI data transfers to all participating nodes.

The BML framework is a thin layer that offers higher-level layers access to multiple BTL modules. It coordinates discovery of peer resources and caches them, and offers a simple round-robin access to all available BTL modules (for striping). When the actual data transfer occurs, the BML layer is bypassed to improve performance [17].

The actual data transfer and connection handling takes place at the BTL. It does not contain any MPI semantics – it just moves bytes across the network. There can be a multitude of btl modules from different BTL components instantiated at a single node, because each network device is associated with its own BTL module.

2.4 Open MPI Startup and Current Address Selection

This section gives a short summary of the way Open MPI exchanges information between processes during startup. Since we are mainly interested in network-related activities, we concentrate on how the exchange of network-specific information and the selection of peer addresses is implemented.

2.4.1 Exchanging Process Connection Information during Startup

The first process started is the so-called head node process (HNP). To be able to share essential run-time information, each process started after the HNP receives a connection string on the command line, which contains information on how to connect to the HNP. It then scans the local environment and opens an out-of-band (OOB) communication channel to the HNP in order to exchange data with the general purpose registry (GPR). The GPR then contains all IP addresses and the associated listening ports of each process, enabling every node to connect to its peers directly. Unless the run-time parameter *mpi_preconnect_all* is set, connections to peer processes are opened lazily, which means that no connection to a peer is established until a message must be transferred. In case of a large message and several transports available, Open MPI is capable of splitting a message and sending over all available interconnects in parallel. This behaviour is referred to as *striping*.

2.4.2 Current Address Selection Method

Algorithm ?? shows the method used by Open MPI to select a peer address used for connection. Note that this method is called several times, until no further peer addresses are available. The algorithm works pretty simple: It iterates over the list of peer addresses, skipping all addresses already in use. If the address is either public IPv4 or IPv6, the address is accepted, marked as 'in use', and returned. If an address is not accepted at this point, it must be a private IPv4 address. In such a case, we iterate over all IP addresses configured on the local node, and if we find a private IPv4

address that belongs to the same network as the peer address, we accept it. If the test fails for all peer addresses, we return `OMPI_ERR_UNREACH`, which denotes that no further connections to the peer are available.

2.4.3 Problems with the Current Approach

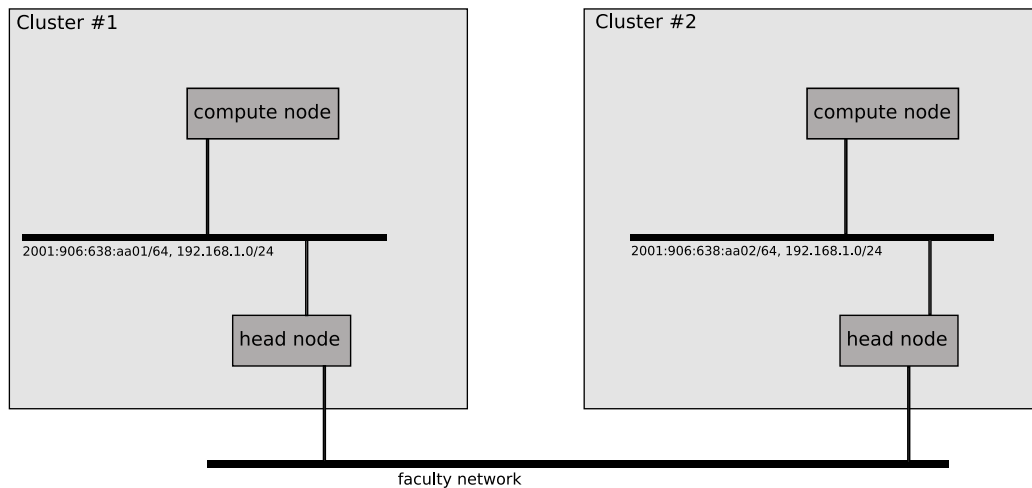


Figure 2.1: A typical multi-domain cluster setup where two clusters are configured using addresses from the same private network.

There are a few minor problems with the current approach. For example, there is no check to assert that the protocol of the selected peer address is also configured on the local node, so if the peer process has a public IPv4 address and IPv6, but the local node is IPv6-only, `mca_btl_tcp_insert` selects either the public IPv4 or the IPv6 address, depending on which it handles first. Since the local node does not have IPv4, connection fails in some cases, even though a transport is available (using IPv6). Consider Figure 2.1: It shows a typical multi-domain cluster setup, where both clusters use a private, unrouted IPv4 network and additionally have public IPv6 configured. When applying Algorithm ??, the RFC1918 address is handled first, and is selected because it appears that both nodes belong to the same network. The following connection attempt either fails or reaches the wrong computer, if the IP address of the peer node also exists in the local network.

These problems can easily be solved by testing for local IPv4 and IPv6 capability before selecting an address of a given family, and by prioritising public addresses over private addresses. However, there are more complicated setups in which the presented method fails to select an optimal

set of peer addresses to maximize throughput. Consider the following table, which depicts two hosts, their network interfaces and the associated IP addresses:

Host	Interface	IPv4 address	IPv6 address
Host A	eth0	193.175.13.2/24	2001:906:638:aa01::02/64
	eth1		2001:906:638:bb01::02/64
Host B	eth0	193.175.14.2/24	
	eth1	193.175.15.2/24	2001:906:638:bb02::02/64

In such a situation, it is possible to utilize both interfaces for striping, if Host A selects the peer addresses 193.175.14.2 and 2001:906:638:bb02::02. However, such a decision can not be made based on only considering a single peer address: At first glance, selecting 193.175.15.2 seems just as good as selecting 193.175.14.2. The first choice would make it infeasible to utilize a second connection though, because Host A already uses interface eth0, and interface eth1 only has IPv6. No connection to the interface eth0 of Host B is thus possible, as it only has an IPv4 address configured.

Since the address selection routine only attempts to connect to a private IPv4 peer address if both nodes reside on the same network, VPN-based solutions or solutions where routing between private networks is properly configured, e.g., inside an administrative domain, will not be able to send messages between certain nodes, because Open MPI simply refuses to attempt a connection.

We observe that the address selection routine currently utilized in Open MPI is not always performing an optimal selection of peer addresses, resulting in a performance loss or, even worse, is not able to establish a connection at all, resulting in a failing application.

Chapter 3

Improved Address Selection

As we have already seen in Section 2.4.3, the address selection algorithm presently used in Open MPI is not always able to handle complicated network setups satisfactory, resulting in a less-than-optimal resource usage or a failing application. In this chapter, we present a new approach which takes the complete network setup into account to find an optimal selection of peer addresses to connect to. In Section 2.1 we identify the goals and requirements for such an algorithm and Section 3.2 discusses the problems arising with private (RFC1918) IPv4 addresses. Based on the discussion and the design goals, in Section 3.3 we show how the address selection problem can be modelled and solved as a graph problem and finally, in Section 3.4 we present an overview of the implementation details.

3.1 Design Goals

The single most important requirement for any address selection routine is to guarantee connectivity. It must be able to identify at least one peer address which is reachable from the local node, and if no such address can be found, the user must be informed and the process aborted.

Open MPI is capable of using several available interconnects in parallel, so throughput can potentially increase when several NICs are available. To exploit this behavior, the address selection routine must be able to identify as many independent links to the peer as possible. However, selecting a set of peer addresses which either connect via the same local interface or to the

same peer interface actually decreases the throughput due to the increased overhead. This situation is called oversubscription and should be avoided for optimal performance.

Determining reachable peer addresses should not rely on probing, i.e. it is not desirable to just connect to a peer process on some address just to see if it is reachable via this address. This is especially important when the peer address in question is a private IPv4 address, because in such a case it is possible to accidentally connect to the wrong computer, e.g., when the compute node we want to contact is part of a different private network which just happens to use addresses from the same address range, and there also exists a computer using the same IP address which is connected to the local network.

Recapitulatory, we have identified three major requirements for the address selection routine:

1. Guarantee connectivity
2. Enable striping while avoiding oversubscription
3. Avoid probing

The requirements presented above might conflict with each other, especially when RFC1918-based address are involved. This is discussed more thoroughly in the following section.

3.2 Connectivity in Private Networks

In this section, we discuss the problems surfacing when private IPv4 addresses are used.

One of the challenges arising when dealing with RFC1918-based private IPv4 address in multi-domain cluster setups is the problem of deciding whether a given peer process is reachable at a given address from the local compute node. If both hosts are connected to the same network, it is safe to use the given private peer address for data transfers. However, multi-domain cluster systems commonly span over several administrative domains, which might not have their address allocation efforts synchronized. This means that a peer address which appears to be from the same

network, based on the IP address and the netmask, can possibly belong to a different network. Without further knowledge and probing, we can not assume connectivity to such an address. On the other hand, it might be possible to successfully establish a connection between addresses belonging to different private networks, e.g. when they belong to the same administrative domain and internal routing is configured accordingly, or when the networks are connected via VPN as discussed in Section 1.2. Again, the information available to the application is not sufficient to make a reasonable decision.

However, Open MPI must be able to handle such situations transparently, preferably without user intervention or manual configuration. To achieve this, a reasonable default behavior must be designed.

We propose the following method to handle RFC1918-based peer addresses which belong to the same network, according to their network prefix and the netmask:

1. Search for address collisions in the GPR
2. If no duplicated IP addresses exist, assume that hosts with the same network prefix actually belong to the same network and a connection is possible.
3. If, on the other hand, address collisions exist, i.e. several hosts export identical IP addresses, forbid the use of private IP addresses for connecting, as reliable connections to hosts residing on networks with colliding address spaces can not be established.

Of course, this is not fail proof: Consider a multi-domain cluster consisting of two independent cluster computers both using the private network 192.168.1.0/24, configured without address collisions across all nodes. The heuristic method above would assume that all nodes belong to the same network, and connections across cluster boundaries would fail with *Destination Host Unreachable*. However, such an exotic setup would not work anyway, leaving no option but to abort the application.

Connections to peer addresses residing on a different private network should generally be disabled, as no assumptions about connectivity can be deduced. However, the Open MPI development mailing list [6] regularly sees bug reports from users attempting to run an application on such a setup, so it might a reasonable course of action to attempt connecting to such an address if no other options are available.

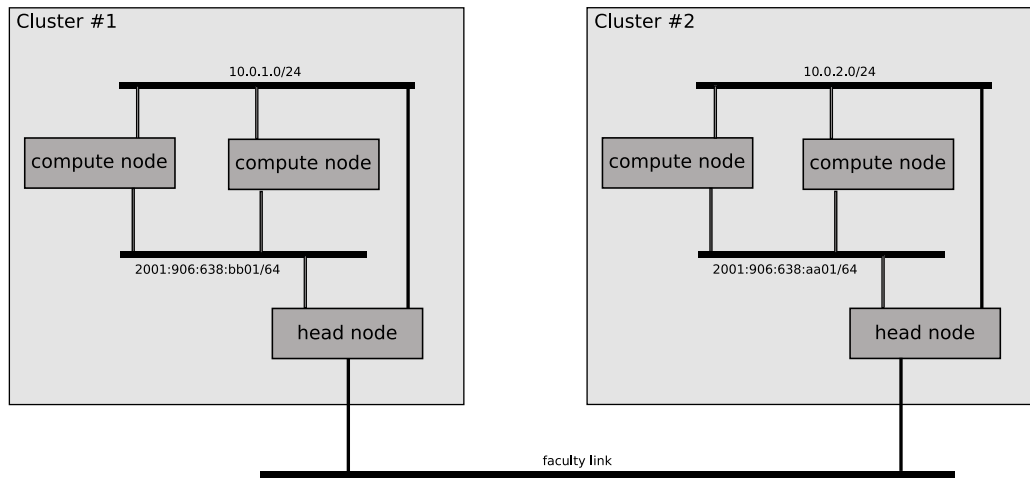


Figure 3.1: A multi-domain cluster system with additional intra-cluster network and a common transport via IPv6

Figure 3.1 illustrates a two-cluster setup with dual internal TCP/IP networks. In this example, striping can be utilized for both clusters' internal communication, because no address duplicates exist and using the private networks is allowed. Messages that cross cluster boundaries are traveling across the faculty network only, because we do not assume that the private addresses from a different network are reachable.

We have seen that there exist a few imponderabilities with private IPv4 addresses, and some of them can not be resolved definitely with the information available to a process. We have to resort to a heuristic to make an educated guess on the underlying network topology.

3.3 Solution to the Address Selection Problem

In this section, we develop a heuristic based on the discussion of design goals from Section 3.1.

Assume that you are given a list of local and peer interfaces with the IP addresses configured on each interface. Selecting a set of peer addresses while respecting the requirements as described in Section 2.1 means to adhere to the following constraints:

1. No pair of peer addresses resides on the same interface on the remote

- node (oversubscribed peer interface).
2. No pair of peer addresses may have their source address on the same local interface (oversubscribed local interface).
 3. A peer address may only be selected if we can be sure that a connection to this address is possible, or there is no other way to reach the peer process.
 4. Connections to peer addresses on the same network as our local addresses should be preferred

For each pair consisting of a local interface and a peer interface, we can determine what kind of connection between these two interfaces is possible, depending on the addresses assigned to the interfaces. Each pair of addresses from the two interfaces defines a possible connection, and among these possible connections we select the one that is most desirable, according to the criteria above (same vs. different network, private vs. public addresses).

This consideration leads to a set of pairs (i, j) , where i is a local interface and j is a remote interface. For each of these pairs, we have a value to qualify the connection type.

3.3.1 Modelling the Network Setup as a Graph

The notation of interface pairs (i, j) closely resembles the definition of a graph. In this section, we use this observation to transform address selection problem into a graph.

edge weight	connection type
0	No connection possible (no common address family, private IPv4 addresses on different networks, or address collisions)
1	Private addresses on the same physical network w/o address collisions
2	Public IPv4 / IPv6 on the same network
3	Public IPv4 / IPv6 on different networks

Table 3.1: Edge weights depending on the connection type

We can model the input as a weighted bipartite graph $G = (U \cup V, E)$, where the vertex set U represents the interfaces identified on the local node, and the vertex set V represents the set of peer interfaces. The edges in E describe the possibility of a connection from a local interface $u \in U$ to a remote interface $v \in V$. If no connection between u and v is possible, e.g., when the interface u only has an IPv4 address configured, and the interface v is IPv6-only, the edge $\{u, v\}$ is assigned a weight of zero, and an integer larger than zero otherwise. To further qualify a connection between a local and a remote interface, the weight associated with an edge depends on what type of connection between that pair of interfaces is possible: Public IPv4 or IPv6 is associated a higher weight than RFC1918 addresses, and address pairs indicating that both interfaces are on the same network are preferred to interfaces where this is not the case. This means that an edge with a higher weight is considered to allow a better connection between to interfaces than an edge with a smaller weight.

Table 3.1 lists the possible connection types and the associated weights. As discussed in Section 3.2, connections to private peer addresses residing on different networks are disabled by default, as are connections to private IPv4 addresses which belong to networks where address duplicates have been detected.

For instance, let us revisit one of the examples from Section 2.4.3, where two hosts A and B had the following addresses configured:

Host	Interface	IPv4 address	IPv6 address
Host A	eth0	193.175.13.2/24	2001:906:638:aa01::02/64
	eth1		2001:906:638:bb01::02/64
Host B	eth0	193.175.14.2/24	
	eth1	193.175.15.2/24	2001:906:638:bb02::02/64

The graph resulting from this network configuration is depicted in Figure 3.2. The vertices *eth0* and *eth1* on the left represent the network interfaces from Host A , while the vertices on the right represent the interfaces of B .

The solution to the address selection problem can now be restated to find a subset of the edges from G , because an edge in G represents a pair of interfaces. To later obtain the peer address resulting from the selected edge, it is sufficient to identify which of the addresses configured on the interface pair resulted in the highest weight.

We have transformed the original input to the address selection problem into a graph, and are able to express our desired solution as a subset of the

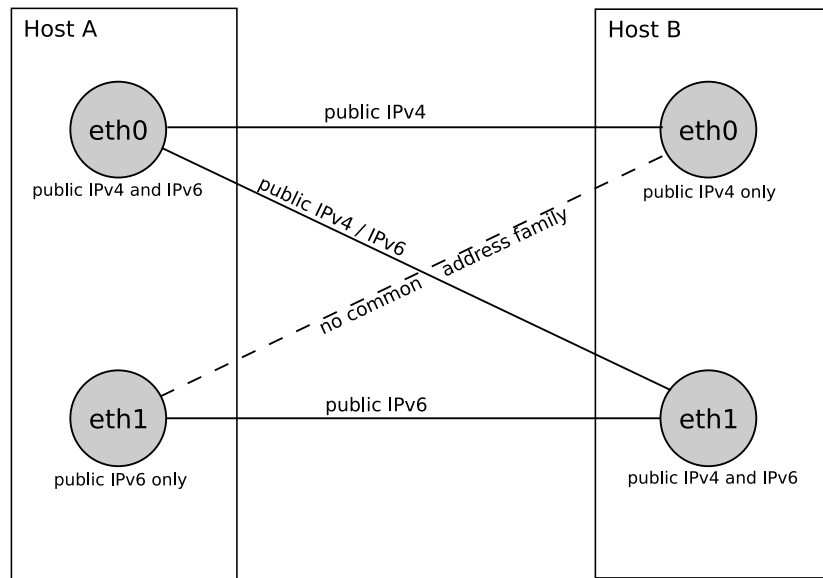


Figure 3.2: The graph resulting from the network configuration presented in Section 2.4.3

edges of the graph.

3.3.2 Properties of a Desired Solution

When reformulating our problem as a graph instance, we also need to express the design goals as discussed in Section 3.1 as graph properties. In this section, we convert the previously informally phrased requirements for a solution of the address selection problem into formal graph language.

To guarantee connectivity, no edges with a weight of zero may be selected. An edge with a weight larger than zero indicates that we are sure that a connection is possible, either because the involved addresses are public IPv4 or IPv6 addresses or . To ensure this property, edges with a weight of zero must be deleted.

To avoid oversubscription, no two selected edges may have a common endpoint on either side, locally or remote. A set of edges meeting this requirement is called a *matching*. It is formally defined as follows:

Definition 1 (Matching) Given a graph $G = (V, E)$, a matching $M \subseteq E$ is a set of pairwise non-adjacent edges, that is, no two edges share a common

endpoint.

To maximize throughput, the number of concurrent connections has to be maximized. This means that we have to maximize the number of edges we select. Such a set of edges is called a *maximum matching*, which is defined in the following way:

Definition 2 (Maximum Matching) *A maximum matching M of a graph $G = (V, E)$ is a matching of G with maximum cardinality, that is, there is no matching M' such that $|M| < |M'|$. Such a matching is also referred to as a maximum cardinality matching.*

Preferring connections over directly attached networks can be achieved by assigning a higher weight to edges if both endpoints share an address of the same network. The maximum cardinality matching described above can be modified to a *maximum cardinality maximum weight matching* (also known as the *assignment problem* to reflect the additional constraints. Please note that this is a problem different from the common *maximum weight matching* problem, because the maximum weight matching simply asks for a matching of maximum weight, independent of its cardinality. In our case, we are searching the matching with maximum weight from the set of all matchings with maximum cardinality.

3.3.3 Solutions for the Assignment Problem

There exist a number of efficient algorithms for the assignment problem: The first published polynomial-time algorithm was Kuhn's classical *hungarian method* [22] with a runtime of $O(m \cdot n^2)$, where m is the number of edges of the input graph G and n is the number of vertices of G . Gabow and Tarjan [13] showed that the problem can be solved in $O(\sqrt{nm} \log(nC))$ under the assumption that the edge weights are integers in the range $[-C, C]$.

All these algorithms have relatively intricate implementations, a typical implementation of the hungarian method in C needs several hundred lines of code. For the application at hand, this effort is hardly justifiable, because the sizes of the instances we need to solve depend on the number of ethernet NICs installed in a compute node. This number is typically very small, even flat neighborhood networks [12] do not have more than 4 ethernet cards installed.

This means that it is not necessary to implement a complex algorithm to solve the assignment problem. Instead, it is sufficient to use a brute-force approach by testing all possible assignments. Even though the number of possible assignments is as large as factorial n , where n is the maximum number of interfaces installed in the local and the remote host, the running time of a brute-force approach is not an issue if both hosts have 6 or less NICs. In fact, due to the high constants involved in the polynomial-time algorithms mentioned above, they are expected to be slower than brute force for typical cluster configurations.

To compute the best selection of peer addresses, we need to be able to compare different selections according to the number of possible concurrent connection and the quality of these connections. We assume that the number of local interfaces is equal to the number of peer interfaces, and insert dummy interfaces if this is not the case. Let n be the number of peer interfaces, and let $P = \{p_1, p_2, \dots, p_n\}$ be a permutation of $1, 2, \dots, n$. Let W be a weight matrix, where w_{ij} defines the weight of a connection from the local interface i to the remote interface j , where a value of 0 indicates that no connection between these interfaces is possible, and higher values indicate that a possible connection between these two interfaces has better properties than a connection with a lower value.

We denote the number of concurrent connections of a permutation P with $C(P)$, and the quality with $Q(P)$. The value $C(P)$ is equal to the number of non-zero entries w_{i,p_i} for $i = 1 \dots n$, and

$$Q(P) = \sum_{i=1}^n w_{ip_i}$$

To find the best set of peer addresses, we have to iterate over all possible permutations P to find the assignment which allows the highest number of concurrent connections. Ties are resolved by selecting the permutation with the higher quality $Q(P)$.

3.3.4 Generating All Permutations

The following algorithm recursively generates all permutations of $1, 2, \dots, n$ in lexicographic order:

The elements of the array a are initialized to zero before the algorithm starts, and `visit()` is initially called with the parameters $k = 0$, $level = -1$ and

size = n . The method `evaluate_assignment()` is called for each generated permutation. Execution time for generating all permutations for $n \leq 7$ is below the precision of measurement even on a Pentium-M running at 600 MHz, which results in no noticeable performance impact because (a) the assignment is calculated at most once per peer process and (b) we have yet to see compute nodes with 7 or more NICs installed.

We have now successfully transformed the address selection problem into a graph problem, and proposed a solution to the reformulated task.

3.4 Implementation

In this section, we give a short summary of the modifications necessary to implement the solution developed in Section 3.3.

Code modifications to implement the new address selection method can be limited to the `btl/tcp` module due to the modular structure of Open MPI. As described in Section 2.4, the method `mca_btl_tcp_proc_insert()` residing in the file `ompi/mca/btl/tcp/btl_tcp_proc.c` is responsible for identifying and assigning one additional peer address each time it is called, until no further connections are possible and `OMPI_ERR_UNREACH` is returned. There is no need to change this behavior, so all changes are performed in that file and the appropriate header file.

We had to implement the following functionalities:

Identify address collisions: We iterate over the process information stored in the GPR, where a **struct** `mca_btl_tcp_proc_t` is stored for each process, which contains an array of all addresses exported by the peer. It is not necessary to compare each exported address with each other address, which would result in a runtime of $O(n^2)$ where n is the number of all addresses exported by all hosts. Instead, we insert all addresses into a hash table, after checking if such an address is not yet stored in the table. If the address already exists in the hash table, we are facing an address duplicate.

This step is executed only once, after all process information is available in the GPR.

Parse the proc structure of the peer we connect to: The **struct** `mca_btl_tcp_proc_t` contains an array of **struct** `mca_btl_tcp_addr_t`, which is defined as follows (for IPv6-enabled Open MPI installations):

```

struct mca_btl_tcp_addr_t {
    struct in6_addr addr_inet; /**< IPv4/IPv6 listen address > */
    in_port_t addr_port; /**< listen port */
    uint16_t addr_ifindex; /**< remote interface index assigned with this address */
    unsigned short addr_inuse; /**< local meaning only */
    uint8_t addr_family; /**< AF_INET or AF_INET6 */
};

```

Even though the data structure contains information about the remote interface the address is configured on (`addr_ifindex`), the address information is not structured in a manageable way. Since our proposed solution is centered around a graph with vertices representing local and remote interfaces, we need to reorganize the data into a suitable interface-centered data structure:

```

struct mca_btl_tcp_interface_t {
    struct sockaddr_storage* ipv4_address;
    struct sockaddr_storage* ipv6_address;
    mca_btl_tcp_addr_t* ipv4_endpoint_addr;
    mca_btl_tcp_addr_t* ipv6_endpoint_addr;
    int ipv4_netmask;
    int ipv6_netmask;
    int kernel_index;
    int index;
    int inuse;
};

```

Listing 3.1: data structure containing all connection and address information per interface

The same data structure is used to store the data provided through the OPAL layer for the local node via `opal_ifbegin()` and `opal_ifnext()`. Note that we do not explicitly build a graph structure: A vertex is simply an integer in the range $0, 1, \dots, n - 1$ where n is the number of local (or peer) interfaces.

Compute weight matrix W : For each pair of local interface i and peer interface j the appropriate entry w_{ij} is computed, depending on the addresses associated with each interface. We use utility functions provided by the OPAL layer, like `opal_net_addr_isipv4public` and `opal_net_samenetwork` to determine whether two addresses belong to the same network and whether they are public IPv4 addresses. Together with the previously computed information about address duplicates, this results in a weight w_{ij} . In addition

to the actual weight we also store the peer address which resulted in that weight in another array of the same dimensions (this will be the peer address selected for connection if the best assignment contains the associated pair of interfaces).

Identify the best interface assignment: As described in Section 3.3.4, we generate each possible permutation P of peer interfaces and calculate the cardinality $C(P)$ and the quality $Q(P)$. We keep track of the permutation with maximum cardinality (and quality, in case of identical cardinalities). Again, we only store an array of integers containing a permutation of $0, 1, \dots, n$ which specifies an assignment.

Select peer address: After determining the best assignment, we iterate over the set of selected peer interfaces to find an interface which is not yet in use. If such an interface exists, we set the appropriate peer address and mark the interface as 'in use', if none can be found (all interfaces are already in use), we return `OMPI_ERR_UNREACH` because no further concurrent connections may be opened.

If no assignment P with cardinality $C(P) > 0$ can be found, this means that no connection to the peer exists with a public IPv4 or IPv6 address, and the two hosts either don't reside on the same private network, or the network is not usable due to address collisions. In such a case, we select a private IPv4 address from the remote host, hoping that routing to this address is properly configured. If no such peer address exists, e.g., the peer is IPv6-only but the local node is IPv4-only, no connection can be established and `OMPI_ERR_UNREACH` is returned.

We implemented the changes as described above in the Open MPI library, the overall size of the modifications is about 500 lines of code.

To sum up this chapter, we developed an improved address selection heuristic by defining a set of design goals for address selection in clusters and multi-domain cluster computers, based on ensuring connectivity and maximizing communication performance. We modelled the problem as a graph, and incorporated the design goals as properties of the solution, which is formed by a subset of the edges of the graph. It turned out that the solution to our address selection problem is a well-known graph problem known as the assignment problem. We chose to solve the problem by a seemingly inferior, exponential time algorithm, but argued that the instances are easily small enough to justify the choice. Finally, we addressed some of the implementation details showing how the Open MPI library was modified to support the improved address selection heuristic.

Chapter 4

Evaluation

In this section, we verify our approach presented in Chapter 3. In Section ??, we introduce our testing framework, which is used to perform all tests on easily configurable cluster computers running on virtual machines. In Section 4.2, we evaluate various cluster setups, ranging from simple clusters to more sophisticated multi-domain cluster systems, to assure connectivity between cluster nodes. Section 4.3 describes the tests performed to guarantee that multiple interconnects are correctly used concurrently to improve performance.

4.1 The Virtual Test Environment

The Virtual Test Environment has been developed as a diploma thesis at the Friedrich-Schiller University of Jena [28]. It provides a framework for setting up and configuring clusters within a few seconds. The framework is used in this work to provide the facilities to easily test our modifications to the Open MPI library on a multitude of cluster systems. It is capable of emulating complex cluster systems and multi-domain cluster computers with arbitrary network topologies using virtual machines. By using a lightweight virtualization technology based on OpenVZ, even large cluster computers consisting of many machines can be simulated on a single workstation to allow continuous, integrated testing. Cluster setups are described directly in Ruby, using syntax elements like `Switch`, `Router`, `Virtual Machine` or `Cluster`. Configuration of a virtual multi-domain cluster computer consisting of two clusters of 16 compute nodes connected via IPv6 can be ex-

pressed in no more than 6 lines of Ruby code, as outlined below:

```
require "vte"  
cluster1 = VTE::Cluster.new 16  
cluster2 = VTE::Cluster.new 16  
switch1 = VTE::Switch.new(:ipv6).connect(c1.vm)  
switch2 = VTE::Switch.new(:ipv6).connect(c2.vm)  
router = VTE::Router.new.connect(s1).connect(s2)
```

The virtual cluster is ready to execute applications within a few seconds, which finally allows continuous testing of distributed applications without the need to have costly hardware available or manually set up or rewire existing hardware to perform the necessary verification of functionality.

Note that VTE does not permit performance analysis of applications, because all virtual machines can reside on a single workstation. For the same reason, testing distributed or parallel applications involving long-taking computations might not be feasible. However, VTE is a powerful tool for testing the communication aspects of your application. Since all network-based communication can also be accessed from the host system, monitoring the network traffic generated by your application from a single point of access is feasible, which is usually not possible when facing a real cluster computer.

4.2 Assuring Connectivity

As we have already noted in Section 3.1, the most important requirement for our address selection routine is to assure connectivity between all participating nodes. In this section, we evaluate the improved address selection heuristic to ensure that Open MPI is able to send MPI messages over a network.

4.2.1 Single Cluster Systems

Even though this work is focused on message passing in multi-domain clusters, Open MPI should be able to handle a wide range of clusters, including typical cluster computers. We evaluate the improved address selection in terms of connectivity for several typical cluster setups.

Private IPv4: The most common interconnect for cluster computers is standard ethernet, using TCP/IP with private IPv4 addresses for each compute node. We create a cluster with 3 nodes and configure a private IPv4 address on each NIC. We then start a simple MPI application which sends a small MPI message (1 B) from node 0 to node 1, which passes it on to node 2 and from there it is sent back to node 0. If node 0 receives the message it originally sent, we can be sure that all nodes are capable of sending and receiving MPI messages, using the improved address selection heuristic.

We execute the following Ruby code in the interactive Ruby interpreter (irb) to configure a virtual cluster consisting of three nodes, which are assigned the IP addresses 192.168.1.2, 192.168.1.3 and 192.168.1.4:

```
require "vte"
cluster = VTE::Cluster.new 3
switch = VTE::Switch.new("192.168.1.0/24").connect(cluster.vm)
p cluster.vm[0].vzid
```

The last statement prints the OpenVZ id of the first virtual machine, which is needed to access the cluster. After all machines are booted, we log into the first virtual machine (vm0) and execute the mpi application “ring” which sends a message around all nodes as described above:

```
peiselt@amun3:~$ sudo vzctl enter 34588
entered into VE 34588
vm0:/# mpirun -host vm0,vm1,vm2 ring
1: waiting for message
2: waiting for message
0: sending message (0) to 1
0: sent message
1: got message (1) from 0, sending to 2
0: got message (2) from 2
2: got message (2) from 1, sending to 0
vm0:/#
```

Note that the output order does not reflect the actual order of execution, because the processes are run in parallel, and ORTE collects and prints the outputs in the order it receives them from the processes. Nevertheless we can see that each process successfully sent and retrieved the message, which means that message passing using the improved address selection heuristic is working in this scenario.

IPv6: We propose IPv6 to establish direct end-to-end connectivity in multi-domain cluster setups. In this experiment, we verify that a single cluster system equipped with IPv6 addresses only is able to exchange MPI messages between its nodes. Analogue to the previous experiment, we run the same MPI application (“ring”), using a modified, IPv6-only cluster setup which is generated as follows:

```
require "vte"  
cluster = VTE::Cluster.new 3  
switch = VTE::Switch.new(:ipv6).connect(cluster.vm)
```

Running the ring test inside one of the virtual machines shows that we successfully sent and received the message at all nodes, so the address selection heuristic developed in this work is able to handle such a cluster setup.

Dual-Stack Setup: During the transition from IPv4 to IPv6, it is quite common to have a setup where a private IPv4 address and an IPv6 address is configured at each NIC, often referred to as a dual-stack setup. To ensure that Open MPI is able to communicate in such a scenario using our heuristic, we check that the ring test is successfully completed on a cluster system generated in the following way:

```
require "vte"  
cluster = VTE::Cluster.new 3  
switch = VTE::Switch.new("192.168.1.0/24").connect(cluster.vm)  
net = VTE::Switch.generate_network(:ipv6)  
cluster.vm.each_with_index {|vm, i| vm.configure_ipv6(net.nth(i+2), "/64")}
```

We have to manually assign the additional IPv6 address to the network interface of each virtual machine. This is done in two steps: At first, we create a network object via `Switch.generate_network(:ipv6)` which reserves a previously unused IPv6 network. Then we iterate over all virtual machines assigned to the cluster `cluster`, and configure an IPv6 address on each of these vms' first ethernet device. To make these addresses unique, the function call `net.nth(i+2)` returns an address of the form `aa:bb:cc:dd::(i+2)`, where `aa:bb:cc:dd` is the network prefix previously generated. The ring test completes successfully on this setup, but it is important to note that we have no information about what IP addresses were selected for communication. Within this experiment, we can only validate that the communication was successful, without further knowledge about how it was performed.

We have seen that the improved address selection heuristic of Open MPI is working in typical single-cluster setups. In the next section, we evaluate whether connectivity can be asserted in various multi-domain cluster setups.

4.2.2 Multi-Domain Clusters

Multi-domain cluster computers are increasingly grabbing attention as they provide a simple way to improve performance by combining existing cluster installations. In this section, we evaluate how Open MPI handles connectivity in typical multi-domain cluster setups when using the improved address selection heuristic.

Routed private IPv4 networks: A typical way to establish end-to-end connectivity between cluster systems is to enable routing between the already-configured private networks. For clusters inside the same administrative domain, internal routing can be configured inside the organizations' network. To connect clusters over a broader distance, a VPN can be used to establish a network route between the cluster systems, as already discussed in Section 1.2.

To verify that Open MPI is able to communicate over cluster boundaries in

such a setup, we create two clusters, each containing only 1 compute node. We apply the ring test to ensure that both compute nodes are able to send a message to their peer. Since we are only interested in the communication between clusters, and not in the communication inside a cluster, limiting the size of the participating clusters to 1 node is sufficient to assert correct behavior, while speeding up the test.

We create two clusters via the following Ruby code:

```
cluster1 = VTE::Cluster.new 1
cluster2 = VTE::Cluster.new 1
switch1 = VTE::Switch.new(:private).connect(cluster1.vm)
switch2 = VTE::Switch.new(:private).connect(cluster2.vm)
router = VTE::Router.new.connect(switch1).connect(switch2)
```

In the last line, the two networks are connected by creating a router between them. Executing the ring test results in a successful message transfer, which means that Open MPI handles such a setup correctly.

Dual-stack setups with private, unrouted IPv4 addresses and IPv6: If no routing between the private networks of the cluster computers can be established, one of the options is to configure IPv6 on all participating sites. If the existing private IPv4 addresses remain, the resulting setup contains clusters where each node has a private IPv4 address and an IPv6 address. The improved address selection heuristic should select the IPv6 addresses to communicate to the peer processes. To verify that the observed behavior matches the expected behavior, we construct a multi-domain cluster where private, unrouted IPv4 addresses are configured, and public IPv6 with appropriate routing is established:

```
cluster1 = VTE::Cluster.new 1
cluster2 = VTE::Cluster.new 1
switch1 = VTE::Switch.new(:ipv6).connect(cluster1.vm)
switch2 = VTE::Switch.new(:ipv6).connect(cluster2.vm)
router = VTE::Router.new.connect(switch1).connect(switch2)
net1 = VTE::Switch.generate_network(:private)
cluster1.vm.each_with_index {|vm, i| vm.configure_ipv4(net1.nth(i+2))}
net2 = VTE::Switch.generate_network(:private)
cluster2.vm.each_with_index {|vm, i| vm.configure_ipv4(net2.nth(i+2))}
```

A connection to the IPv4 address of the peer node would fail, as the destination address is unreachable. The ring test succeeds, so we conclude that IPv6 was used for communication, which is what we expected.

The improved address selection routine of Open MPI correctly handles inter-cluster communication in both scenarios, and we have already seen that connectivity in typical single-cluster systems is also established.

4.3 Striping

Open MPI is capable of using multiple interconnects concurrently to maximize bandwidth. The improved address selection heuristic must be able to select a set of peer address to be used for communication, according to the design rules as discussed in Section 3.1.

Single cluster with 2 NICs per host: If the compute nodes are equipped with 2 ethernet cards, Open MPI splits large messages into small parts and transfers them over the available transports in parallel. The network interfaces of the virtual machines can be monitored to verify that all available transports have been used. We create a cluster consisting of 2 nodes, and create an MPI application that sends a large message from node 0 to node 1, and, upon successfully receiving the message, back to node 0. The process running on node 0 verifies that the received message matches what was previously sent. Before starting the message transfer, we start `tcpdump` on each interface to monitor the network traffic generated by our application. If message striping was successful, we expect that the amount of data transferred over all interfaces is almost equally split over the interfaces.

Using VTE, additional NICs are easily added to a virtual machine:

```
cluster = VTE::Cluster.new 2
switch1 = VTE::Switch.new("192.168.17.0/24").connect(cluster.vm)
switch2 = VTE::Switch.new("192.168.18.0/24").connect(cluster.vm,1)
```

Note that the second switch (`switch2`) connects the same nodes as the first switch (`switch1`), but is called with an additional parameter, 1: it denotes that the switch connects the virtual machines via `eth1`, the default value for the parameter is 0, so when it is omitted `eth0` is used.

After setting up the cluster and its network, we open three sessions in the first virtual machine, `vm0`: Two sessions are used to monitor the network devices `eth0` and `eth1` by running `tcpdump -i <interface>`, the third session is used to start the application.

We verify that the application successfully transferred the message, and

stop the tcpdump instance to check the output:

```
vm0:/# tcpdump -i eth0
<lots of output>
35661 packets captured
35661 packets received by filter
0 packets dropped by kernel
```

```
vm0:/# tcpdump -i eth1
<lots of output>
35435 packets captured
35435 packets received by filter
0 packets dropped by kernel
```

We note that the number of packets transferred on eth0 is slightly larger than the number of packets captured on eth1. This can be explained by the fact that we captured all traffic, including traffic generated by process startup (ORTE uses ssh in our scenario) and out-of-band communication. The numbers show, however, that both interfaces were utilized by Open MPI to transfer the data, and message striping was applied during our test.

Our tests indicate that the improved address selection heuristic handles many cluster scenarios correctly. However, such tests can never cover all possible cluster setups, and we concentrated on what we believe how typical cluster installations are configured.

Chapter 5

Summary

We improved the address selection heuristic previously used in Open MPI, to deal with new challenges arising in the field of multi-domain cluster systems and the resulting increased complexity in the network topologies message passing systems must be able to handle to achieve the desired connectivity. Our approach also takes the requirements of single-cluster installations into account, as one of the goals of the Open MPI project is to provide a message passing library for all kinds of systems, ranging from small clusters to large, distributed systems spanning several administrative domains. Another important requirement is the full utilization of network resources available by concurrently using all available transports for message passing, without reducing the efficiency by oversubscribing. Our heuristic selects the maximum number of interconnects, while making sure that efficiency in message passing is not reduced by oversubscribing NICs.

We evaluated our approach to ensure that connectivity is guaranteed under various single and multi-domain cluster setups, and additionally verified that message striping is successfully applied.

The presented, improved address selection enables distributed, multi-domain cluster computers to work out of the box, taking away the burden of manual intervention to enable direct end-to-end connectivity between participating compute nodes. Even though IPv6 is not widely deployed at the moment, there are serious efforts, especially from academic and government institutions, to fully enable IPv6 on the internet.

There is still a lot of work to be done in the realm of multi-domain cluster computing: Hierarchical network topologies consist of fast, switched

interconnects inside cluster systems, and slower, wide-area connections as bottlenecks between these systems. Awareness of such topology issues enables faster handling of collective operations and might reduce the traffic on the inter-cluster links. To make this effective, a first step is to generate a global view of the network the application is executed on. In a second step, collective operations must make use of this information by adapting their communication scheme to the identified topology.

Bibliography

- [1] Das native IPv6-Netzwerk des DFN. <http://www.6win.de/>.
- [2] General highlights from the top 500 since the last edition. <http://www.top500.org/lists/2007/06/highlights/general>.
- [3] Myrinet website. <http://www.myricom.com/myrinet/overview/>.
- [4] Open MPI. <http://www.open-mpi.org>.
- [5] Open MPI frameworks. <http://www.open-mpi.org/faq/?category=tuning#frameworks>.
- [6] Open MPI mailing list. <http://www.open-mpi.org/community/lists/ompi.php>.
- [7] Top500 supercomputer sites. <http://www.top500.org>.
- [8] *Fault Tolerant Communication Library and Applications for High Performance Computing*, Santa Fe, NM, 2003. Proceedings of the Los Alamos Computer Science Institute Symposium.
- [9] Olivier Aumage. Heterogeneous multi-cluster networking with the Madeleine III communication library. *IPDPS*, page 85b, 2002.
- [10] Gregory D. Burns, Raja B. Daoud, and James R. Vaigl. Lam: An open cluster environment for mpi. In *Supercomputing Symposium '94*, Toronto, Canada, June 1994.
- [11] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G .E. Fagg. The open run-time environment (openrte): A transparent multi-cluster environment for high-performance computing. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

- [12] H. Dietz and T. Mattox. Klat2's flat neighborhood network, 2000.
- [13] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.
- [14] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kam-badur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
- [15] Edgar Gabriel, Michael M. Resch, Thomas Beisel, and Rainer Keller. Distributed computing in a heterogeneous computing environment. In *EuroPVM/MPI*, pages 180–187, 1998.
- [16] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. *Int. J. Parallel Program.*, 31(4):285–303, 2003.
- [17] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.
- [18] Christian Kauhaus and Dietmar Fey. Building Mini-Grid environments with Virtual Private Networks: A pragmatic approach. In *Proc. PAR-ELEC, Bialystok, Poland, September 13–17*, pages 111–115, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
- [19] Christian Kauhaus, Adrian Knoth, Thomas Peiselt, and Dietmar Fey. Efficient message passing on multi-clusters: An ipv6 extension to Open MPI. In *Proceedings of KiCC'07, Chemnitzer Informatik Berichte*, February 2007.
- [20] Adrian Knoth. IPv6-Message-passing mit Open MPI. 2007.
- [21] Andreas Koch. High performance computing cluster. 2003.
- [22] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, pages 83–97, 1955.

- [23] Rafael Martínez-Torres. PVM-3.4.4 + IPv6: Full grid connectivity. In *EuroPVM/MPI*, pages 233–240, 2005.
- [24] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.
- [25] Mario Petrone and Roberto Zarrelli. Enabling pvm to build parallel multidomain virtual machines. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 187–194, Washington, DC, USA, 2006. IEEE Computer Society.
- [26] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996.
- [27] Lars Schneidenbach and Bettina Schnor. Migration of MPI applications to IPv6 networks. In T. Fahringer and M. H. Hamza, editors, *Proc. Parallel and Distributed Computing and Networks (PDCN 2005)*, Calgary, 2005. ACTA Press.
- [28] Wolfgang Schnerring. Testen von verteilten Systemen in heterogenen Netzwerken mit virtuellen Maschinen am Beispiel von MPI. 2007.
- [29] Steve Traugott and Joel Huddleston. Bootstrapping an infrastructure. In *Proceedings of the Twelfth USENIX Systems Administration Conference (LISA '98), December 6–11*, Boston, Massachusetts, USA, 1998. <http://www.infrastructures.org/papers/bootstrap/>.
- [30] T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 303–310, Budapest, Hungary, September 2004.

Appendix A

CD contents

Location	Description
ompi/	The modified Open MPI library including the improved address selection heuristic
test/	All integrated tests and the VTE library
diplomarbeit/	Diploma thesis including all files used to generate this document

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, 30. September 2007

Thomas Peiselt