Design and Implementation of Scalable High-performance Communication Libraries for Wide-area Computing Environments (広域計算環境における並列計算用のスケーラブルな高性能通信ライブラリの設計と実装)

斎藤秀雄

Abstract

Over the past ten years, clusters have become the predominant architecture for performing parallel computation. By connecting multiple compute nodes by a Local Area Network (LAN), clusters make a large amount of processing power, memory and storage available for parallel computation. By connecting two or more of these clusters by a Wide Area Network (WAN), even more computational resources become available for parallel computation. Recently, the bandwidth of WANs has increased significantly, increasing the number of applications that can potentially take advantage of multi-cluster environments. Unfortunately, multi-cluster environments are significantly more complex than single cluster environments. In particular, they introduce or magnify problems concerning connectivity, scalability, locality and adaptivity. As it is undesirable and unrealistic for each individual application to handle these problems separately, demands have increased for wide-area communication libraries that handle these problems.

Concerning connectivity, wide-area communication libraries need to be aware that connections between clusters are commonly blocked by firewalls or Network Address Translation (NAT). A simplistic scheme that assumes that all processes can connect to each other will encounter problems when deployed in WANs. Only some connections will be allowed, and messages must be routed between every pair of processes using those connections. As for scalability, wide-area communication libraries need to avoid simplistic schemes that establish a large number of connections. While all connections consume resources, wide-area connections especially consume a lot of resources, causing various resource allocation problems. In addition to resource allocation problems, using a large number of wide-area connections in an uncoordinated fashion can result in low communication performance due to congestion. The two previous requirements basically say that a wide-area communication library will establish connections between a subset of all process pairs. Then in order to maintain high communication performance, the process pairs that do establish connections should be selected in a locality-aware manner. In general, connections between nearby processes should be favored over connections between faraway processes, and connections between processes that communicate frequently should be favored over those that communicate infrequently. Moreover, wide-area communication libraries should automatically satisfy the three previous requirements by adapting to environments and to applications. They should not rely on manual configuration, because it is tedious, it does not scale, and it is the cause of various errors.

Much previous research has focused on each of these requirements separately, but more work is necessary in order for wide-area communication libraries to meet all of these requirements at the same time. For example, research centered around message passing offers good locality but poor scalability and adaptivity, while research centered around Peer-to-Peer (P2P) overlay networks offers good scalability and adaptivity but poor locality. This has motivated me to make two proposals concerning the design and implementation of scalable high-performance communication libraries for wide-area computing environments: a locality-aware connection management scheme and a locality-aware rank assignment scheme.

My connection management scheme overcomes firewalls and NAT by constructing an overlay network, and achieves scalability by limiting the number of connections that each process establishes to $O(\log n)$ when the total number of processes is n. In order to achieve high performance with a limited number of connections, the connections that are established are selected in a locality-aware manner, based on latency and traffic information obtained from a short profiling run. Meanwhile, my rank assignment scheme finds a low-overhead mapping between ranks and processes by formulating the rank assignment problem as a Quadratic Assignment Problem (QAP). It adapts to environments and to applications by setting up QAPs using the latency and traffic information obtained from the profiling run.

Using the proposed connection management and rank assignment schemes, I have implemented a wide-area Message Passing Interface (MPI) library called Multi-Cluster MPI (MC-MPI). I have evaluated its performance using the NAS Parallel Benchmarks (NPB) and Distributed Verification Environment (DiVinE). Using 256 cores distributed equally across 4 clusters, MC-MPI was able to limit the percentage of connections that each process established to as low as 10 percent, while performing at least as well as existing methods. For benchmarks in which many processes communicated simultaneously, MC-MPI actually performed better than when many connections were used. Moreover, MC-MPI was able to find rank assignments that performed up to 1.2 times better than commonly used assignments based on host names and up to 4.0 times better than locality-unaware assignments.

In order to support not just MPI applications but any parallel application, I have also used the proposed connection management scheme to implement a wide-area Sockets library called

Scalable Sockets (SSOCK). In a 13-cluster environment with firewalls and NAT, SSOCK was able to connect 1,262 processes with each other without any of the connectivity issues and resource allocation problems that were encountered by existing methods. In another experiment in which many processes simultaneously tried to establish connections with each other, SSOCK was able to quickly establish connections between all pairs of processes, while an existing method suffered from a large number of packet losses and finally timed out. Moreover, the point-to-point communication performance of SSOCK was comparable to that of an existing method, while the collective communication performance was better.

Acknowledgements

First, I would like to express my sincere gratitude to the members of my dissertation committee. Professors Masaru Kitsuregawa and Masahiro Goshima gave me invaluable comments at our annual advisor meetings, and Professors Kei Hiraki, Shuichi Sakai and Hiroyuki Morikawa gave me important suggestions regarding the presentation of my work. It is because of their help that my dissertation is in one piece.

I am deeply indebted to Professors Takashi Chikayama and Kenjiro Taura. As boss of our lab, Professor Chikayama was my most important role model. He gave me some of the most valuable suggestions, and made sure that I had the resources necessary for my research. Meanwhile, Professor Taura was the one who initially got me interested in research. Then as my supervisor for six years, he kept me motivated and led me through three degrees.

I am grateful to everyone in my lab for helping me finish this dissertation on time. Daisaku Yokoyama and Yoshikazu Kamoshida gave me many insightful comments on my work. Tatsuya Shirai, Kei Takahashi and Ken Hironaka procrastinated with me before numerous deadlines. They also worked on some grueling projects with me—we deserve extra degrees for our fast Flowshop solver and for setting up so many clusters!

Finally, a huge thanks goes to everyone else who supported me in various other ways as I worked through my dissertation. In particular, I would like to thank Takuro Matsuda, Yuichi Sei and Chinatsu Nakamura for their friendship, and my parents for sending me through countless years of school.

December 17, 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

Over the past ten years, clusters have become the predominant architecture for performing parallel computation. Over 80 percent of the systems on the November 2008 version of the TOP500 Supercomputing Sites list are clusters, while fewer than 1 percent were clusters on the November 1998 list [64]. By connecting multiple compute nodes by a Local Area Network (LAN), clusters make a large amount of processing power, memory and storage available for parallel computation. By connecting two or more of these clusters by a Wide Area Network (WAN), even more computational resources become available for parallel computation. Message passing applications, data intensive applications, and applications that use distributed file systems are just a few of the applications that can take advantage of these resources.

Until recently, the narrow bandwidth of WANs prevented many of these applications from being executed efficiently across multiple clusters. Recently, however, the bandwidth of WANs has increased significantly. For example, the Science Information NETwork 3 (SINET3), which connects universities and research institutions in Japan, has a capacity of up to 40Gbps [48]. Surfnet, a similar network in the Netherlands, has a capacity of up to 10Gbps [69]. These fast WANs act as backbones for multi-cluster platforms such as InTrigger in Japan [32], the Distributed ASCI Supercomputer 3 (DAS-3) in the Netherlands [74] and Grid'5000 in France [21], and has increased the number of applications that can potentially take advantage of multi-cluster environments.

Unfortunately, multi-cluster environments are significantly more complex than single cluster environments. In particular, they introduce or magnify problems concerning connectivity, scalability and locality. As it is undesirable and unrealistic for each individual application to

Table 1.1: Number of concurrent sessions that some common firewalls can handle (throughput is also given as a measure of the scale of the firewall)

Firewall	Concurrent sessions	Throughput
WatchGuard Firebox®Edge X55e	10,000	100 Mbps
Juniper Networks NetScreen-208	128,000	375 Mbps
ITOS CR1000i	400,000	1 Gbps
Fortinet FortiGate-3600	1,000,000	4 Gbps

handle these problems separately, demands have increased for communication libraries that are wide-area-enabled. Such wide-area communication libraries should satisfy the following requirements:

Connectivity: Wide-area communication libraries need to be aware that connections between clusters are commonly blocked by firewalls or Network Address Translation (NAT) [66]. A simplistic scheme that assumes that all processes can connect to each other will encounter problems when deployed in WANs. Only some connections will be allowed, and messages must be routed between every pair of processes using those connections.

Scalability: In order to scale to a large number of processes, wide-area communication libraries need to avoid simplistic schemes that establish a large number of connections. While all connections consume resources, wide-area connections especially consume a lot of resources, causing various resource allocation problems. For example, the number of sessions that a NAT gateway can handle is limited to about 65,000 (the number of ports), and the number of sessions that a stateful firewall can handle is also limited (Table 1.1 lists the number of concurrent sessions that some common firewalls can handle). These limitations apply to communication using the User Datagram Protocol (UDP) [54] as well as that using the Transmission Control Protocol (TCP) [55], because NAT gateways and stateful firewalls remember states for both protocols. Another resource consumed by each connection is memory. Wide-area TCP connections especially use a large amount of memory, because the send and receive buffers of a connection must be at least as large as the bandwidth-delay product in order to achieve high bandwidth [65]. In addition to resource allocation problems, using a large number of wide-area connections simultaneously in an uncoordinated fashion can result in low communication performance due to congestion.

Locality: The two previous requirements basically say that a wide-area communication library will establish connections between a subset of all process pairs. Then in order to maintain high communication performance, the process pairs that do establish connections should be selected in a locality-aware manner. In general, connections between nearby processes should be favored over those between faraway processes (environment awareness). Moreover, connections between processes that communicate frequently should be favored over those that communicate infrequently (application awareness).

Adaptivity: Wide-area communication libraries should satisfy the three previous requirements automatically by adapting to environments and to applications. They should not rely on manual configuration, because it is tedious, it does not scale, and it is the cause of various errors (improper configuration can result in poor performance or even loss of connectivity).

Much previous research has focused on each of these requirements separately, but more work is necessary in order for wide-area communication libraries to meet all of these requirements at the same time. For example, research centered around message passing offers good locality but poor scalability and adaptivity [22, 30, 31, 33, 34, 37, 46, 70, 71], while research centered around Peer-to-Peer (P2P) overlay networks offers good scalability and adaptivity but poor locality [63, 68, 81]. This has motivated me to study the design and implementation of scalable high-performance communication libraries for wide-area computing environments.

1.2 Contributions

The main contributions of this dissertation are as follows:

1. **Locality-aware connection management**

   I propose a connection management scheme for wide-area communication libraries. This scheme overcomes firewalls and NAT by constructing an overlay network, and achieves scalability by limiting the number of connections that each process establishes to $O(\log n)$ when the total number of processes is n. In order to achieve high performance with a limited number of connections, the connections that are established are selected in a locality-aware manner, based on latency and traffic information obtained from a short profiling run.

2. **Locality-aware rank assignment**

   I also propose a rank assignment scheme for wide-area communication libraries. This scheme finds a low-overhead mapping between ranks (process IDs) and processes by formulating the rank assignment problem as a Quadratic Assignment Problem (QAP) [38]. It adapts to environments and to applications by setting up QAPs using latency and traffic information obtained from the profiling run.

3. **Multi-Cluster MPI (MC-MPI)**

   Using the proposed connection management and rank assignment schemes, I have implemented a wide-area Message Passing Interface (MPI) [42] library called Multi-Cluster MPI (MC-MPI). I have evaluated its performance using the NAS Parallel Benchmarks (NPB) [13, 75] and Distributed Verification Environment (DiVinE) [2, 28]. Using 256 cores distributed equally across 4 clusters, MC-MPI was able to limit the percentage of connections that each process established to as low as 10 percent, while performing at least as well as existing methods. For benchmarks in which many processes communicated simultaneously, MC-MPI actually performed better than when many connections were used. Moreover, MC-MPI was able to find rank assignments that performed up to 1.2 times better than commonly used assignments based on host names and up to 4.0 times better than locality-unaware assignments.

4. **Scalable Sockets (SSOCK)**

   In order to support not just MPI applications but any parallel application, I have used my connection management scheme to implement a wide-area Sockets [53] library called Scalable Sockets (SSOCK). In a 13-cluster environment with firewalls and NAT, SSOCK was able to connect 1,262 processes with each other without any of the connectivity issues and resource allocation problems that were encountered by existing methods. In another experiment in which many processes simultaneously tried to establish connections with each other, SSOCK was able to quickly establish connections between all pairs of processes, while an existing method suffered from a large number of packet losses and finally timed out. Moreover, the point-to-point communication performance of SSOCK was comparable to that of an existing method, while the collective communication performance was better.

1.3 Organization of this dissertation

The rest of this dissertation is organized as follows. First, in Chapter 2, I discuss related work in the general context. Then, in Chapter 3, I discuss work specific to message passing, and give a brief overview of the message passing model itself. In Chapters 4 and 5, I describe the design of MC-MPI, explain my connection management and rank assignment schemes, and discuss relevant experimental results. In Chapter 6, I describe the design and implementation of SSOCK, and discuss relevant experimental results. Finally, in Chapter 7, I present my concluding remarks and possible future directions of research.

Chapter 2

Related Work

2.1 Overview

This chapter discusses related work in a general context. Work specific to message passing, along with a brief overview of the message passing programming model itself, is discussed in Chapter 3.

2.2 VPN

One way to get rid of the connectivity problems arising from firewalls and Network Address Translation (NAT) [66] is to use a Virtual Private Network (VPN).

IP-VPN is a class of VPNs in which the Internet Service Provider (ISP) constructs a VPN. A common method of constructing an IP-VPN is to use BGP/MPLS VPN [60]. This method uses extensions to the Border Gateway Protocol (BGP) [58] to construct multiple VPNs within a single Multi Protocol Label Switching (MPLS) [61] network. The main problem with IP-VPN is that it is inflexible; joining and leaving the VPN can only happen on a per cluster basis, requires changing IP addresses, and requires intervention of the ISP.

VPNs can also be constructed at the user level, using software such as OpenVPN [50] and PacketiX VPN [29]. These solutions virtualize Ethernet [39] by providing tap devices that act as virtual Ethernet adapters and bridge devices that act as virtual Ethernet hubs. Although user-level VPNs are much more flexible than IP-VPN, they still require some manual configuration, and they do not solve the scalability problems involved with establishing a large number of connections.

2.3 Firewall and NAT traversal techniques

Transmission Control Protocol (TCP) splicing is a method for traversing firewalls and NAT [12, 25]. This method allows two nodes to connect to each other even if both nodes are behind firewalls that deny incoming connection attempts. Focusing on the fact that a typical firewall permits replies to outgoing traffic, two nodes repeatedly send SYN packets to each other in hope that the firewall on the other end will let one of the packets pass through. When one or both machines are behind a NAT gateway, the connecting node must know which port will be used by the gateway on the other end, but most NAT implementations use predictable mapping schemes [24]. For the User Datagram Protocol (UDP), there is a similar method known as UDP hole punching [62]. Unfortunately, TCP splicing and UDP hole punching only work in some common situations. For example, they do not work when firewalls have complicated filtering rules or when NAT is used inside a network already using NAT. Thus, these techniques alone cannot solve all of the connectivity problems in a complex wide-area computing environment.

Another way to traverse firewalls and NAT is to insert helper agents into the network. Proxy servers, such as SOCKS [40], and Application Level Gateways (ALGs) [67] fall into this category. These agents solve connectivity problems by performing communication in place of nodes that cannot communicate directly. However, some nodes may need the help of multiple agents in order to communicate, and manual configuration to allow a large number of nodes to communicate all-to-all can become quite a difficult task. Moreover, these agents solve connectivity issues but not scalability issues; connecting a large number of nodes using proxy servers or ALGs will result in a large number of wide-area connections.

2.4 SmartSockets

Maassen et al. have developed SmartSockets [41], a Java library that allows programmers to use sockets in wide-area environments without worrying about connectivity issues. It provides extended versions of the `Socket` and `ServerSocket` classes that transparently connect normally unconnectable nodes by selecting the proper technique from the following set of techniques:

Direct Establishes a normal connection from the source to the destination. This technique only works when communication is not blocked by firewalls or NAT.

Reverse Establishes a connection in the reverse direction, from the destination to the source.

This technique works if the source is not behind a firewall or NAT, even if the destination is.

Splicing Uses the firewall and NAT traversal technique explained in Section 2.3.

Routed Uses forwarding daemons to route messages between nodes that cannot be connected directly.

Unlike proxy servers and ALGs, SmartSockets involves almost no configuration. It automatically chooses the correct technique, and the routing tables used by the forwarding daemons are constructed automatically as well.

SmartSockets is the closest work to SSOCK (Chapter 6), but the main difference is that SSOCK not only deals with connectivity issues but also with scalability issues. SmartSockets establishes direct connections whenever possible, but SSOCK avoids establishing certain connections even when possible.

2.5 P2P overlay networks and IPOP

One way of connecting a large number of nodes in a scalable fashion is to use a Peer-to-Peer (P2P) overlay network. P2P overlay networks achieve scalability by limiting the number of nodes with which each node communicates, and a routing layer forwards messages for nodes that do not communicate directly. Some representative P2P overlay networks include Content Addressable Network (CAN) [57], Chord [68], Pastry [63] and Tapestry [81]. In CAN, n nodes are assigned IDs in a d-dimensional ID coordinate space, and nodes can communicate with each other with an average of $O(dn^{\frac{1}{d}})$ hops if each node maintains information about just $O(d)$ other nodes. In Chord, nodes are assigned IDs in a circular ID space, and nodes can communicate with each other with an average of $O(\log n)$ hops if each node maintains information about $O(\log n)$ other nodes. Pastry [63] and Tapestry [81], which use a variant of Plaxton's distributed search technique [52], also allow n nodes to communicate with each other with $O(\log n)$ hops when each node maintains information about $O(\log n)$ other nodes. The problem with P2P overlay networks is that they make routing decisions based on addresses assigned in the P2P address space, which often causes messages to take long detours in the physical network.

Ganguly et al. have proposed IP over P2P (IPOP) [17], which performs IP tunneling using Brunet [7], a P2P overlay network that works in environments with firewalls and NAT. In order to overcome the previously mentioned detour problem of P2P overlay networks, Ganguly et al. have also proposed to establish shortcut connections between nodes that communicate frequently [18]. However, this method does not work well for applications in which many node pairs communicate frequently, because it will result in a large number of shortcut connections. Moreover, the virtual interface that IPOP uses to perform IP tunneling has an overhead that is unacceptable for many parallel applications; while the latency within a cluster is several to several tens of microseconds, the overhead incurred by the virtual interface is several milliseconds.

2.6 Adaptive routing in wireless networks

In many ways, adaptive routing schemes used in wireless networks [11, 15, 19, 78] are similar to my connection management scheme. They assume that connectivity is limited, perform link quality measurements, and recognize that shortest path routing does not necessarily result in the best performance. However, there are some major differences between the two. For example, signal stability is an important issue in wireless networks, but not in wired networks used for parallel computation. Meanwhile, unlike my connection management scheme, routing schemes for wireless networks do not need to handle the problem of reducing wide-area connections; faraway nodes can often communicate in wired networks (except when firewalls and NAT prevent it), but only nearby nodes can communicate in wireless networks.

Chapter 3

Message Passing

3.1 Overview

Message passing is a parallel programming model for distributed memory systems, and is commonly used for parallel programming on clusters. In message passing, communication is programmed explicitly. As a result, message passing usually performs better than other parallel programming models such as distributed shared memory (DSM), but is often harder to write.

The Sockets Application Programming Interface (API) [53], in which communication is programmed using the `send` and `recv` function calls, is the most basic API for message passing programming. The Sockets API does not provide much abstraction in terms of parallel programming, so it is not too common for parallel applications to be written using this API. However, middleware for parallel and distributed systems (e.g., communication libraries and distributed file systems) are often written using this API. Thus, improving the Sockets API can benefit parallel applications indirectly. While this is not the main focus of this dissertation, I briefly discuss how my work can be applied to the Sockets API in Chapter 6.

The Message Passing Interface (MPI) [42], the main focus of this dissertation, is the de facto standard API for message passing programming. It provides a higher level of abstraction than the Sockets API by assigning a globally unique identifier (*rank*) to each process. The underlying message passing system assigns a rank to each process inside an initialization function, and communication patterns are expressed in terms of ranks and the total number of processes. This makes the task of translating parallel algorithms to message passing programs intuitive. The APIs for initialization, obtaining the rank of a process, and obtaining the total number of processes is as follows:

- `MPI_Init(int *argc, char **argv);`

(initializes the MPI execution environment)

- `MPI_Comm_rank(MPI_Comm comm, int *rank);`
  (returns the rank of the calling process in `rank`)

- `MPI_Comm_size(MPI_Comm comm, int *size);`
  (returns the total number of processes in `size`)

Another feature of MPI that makes it suitable for parallel programming is that the API is platform independent. Platform-dependent issues are handled by the message passing system, not the application. This helps message passing programs to be portable across different architectures and even to work in heterogeneous environments.

The rest of this chapter is organized as follows. First, in Sections 3.2 and 3.3, I describe the two main kinds of communication primitives provided by MPI, point-to-point communication and collective communication. Then, in Section 3.4, I describe the issues involved with running MPI in wide-area environments. In Section 3.5, I describe the need for connection management and the difficulties involved with it. Finally, in Section 3.6, I discuss different rank assignment techniques for lowering communication overhead.

3.2 Point-to-point communication

Point-to-point operations, or 1-to-1 operations (i.e., send and receive), are used for communication between two processes. While there are many variants with different behaviors and semantics, the two most basic point-to-point operations are the following:

- `MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);`
  (performs a blocking send to the process with rank `dest`)

- `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);`
  (performs a blocking receive from the process with rank `source`)

3.3 Collective communication

Collective operations are communication operations that involve a group of processes. They can be divided into three different groups based on the direction of data flow: 1-to-N operations (e.g., broadcast), N-to-1 operations (e.g., reduction) and N-to-N operations (e.g., all-to-all). Some specific examples include the following:

- `MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`
  (broadcasts a message from the process with rank `root` to all other processes)

- `MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);`
  (reduces values on all processes to a single value on the process with rank `root`)

- `MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm);`
  (sends data from all to all processes)

Programming with collective operations is much easier than with just point-to-point operations, because collective operations can provide much functionality in just one function call. At the same time, the use of collective operations can also improve the performance of parallel programs. This is because message passing systems can take advantage of information such as bandwidth and latency to provide optimal collective operations for a given environment. Gorlatch provides an in-detail discussion on the effectiveness of collective operations in [20].

Proposing specific algorithms for performing collective operations efficiently is beyond the scope of this dissertation. However, the heavy traffic caused by collective operations is very relevant to connection management. This is discussed in more detail in Section 3.5.

3.4 Wide-area-enabled MPI libraries

While MPI has mostly been used for parallel computation on single clusters, there are strong demands for running MPI applications across multiple clusters. Multi-cluster environments are appealing, because they offer much more computational resources than single cluster environments. Unfortunately, multi-cluster environments are significantly more complex than single cluster environments, introducing or magnifying problems concerning connectivity, scalability and locality.

For example, collective communication algorithms for single cluster environments perform poorly in multi-cluster environments. These algorithms often assume that the network is homogeneous [73], but such an assumption causes messages to go back and forth between clusters when used in multi-cluster environments. Thus, some wide-area-enabled MPI libraries, including MagPIe [37] and MPICH-G2 [33, 34, 46], provide network-heterogeneity-aware collective operations. These libraries avoid messages from going back and forth between clusters by differentiating between intra-cluster and inter-cluster communication and performing communication hierarchically.

Another issue in multi-cluster environments is connectivity. Some wide-area-enabled MPI libraries have forwarding mechanisms that allow all processes to communicate with each other, even in the presence of firewalls and Network Address Translation (NAT). GridMPI [22,70,71], MPICH/MADIII [1] and PACX-MPI [16] fall into this category. These libraries use manually provided information to connect some processes directly and establish multi-hop routes between other processes. The problem with this approach is that the amount of necessary configuration can become overwhelming as more computational resources and more complex environments are used. Moreover, manual configuration is static, but factors such as load and faults change the resources available for a computation from execution to execution. MC-MPI (Chapter 4) differs from these libraries in that it does not require static, manual configuration; it discovers process connectivity automatically.

3.5 Connection management

MPI libraries typically deliver point-to-point messages directly, and only route messages through other processes when firewalls or NAT prevent direct communication. This is because direct communication usually has lower latency and higher bandwidth than routed communication. A simplistic implementation that delivers messages directly would pre-establish connections between all pairs of processes inside `MPI_Init`. However, such a design does not scale well, because it results in a large number of connections. Moreover, when many process pairs communicate simultaneously, using a large number of connections results in low communication performance due to congestion. These problems are especially significant when there is a large number of wide-area connections, because wide-area connections consume more resources and are more prone to congestion than local-area connections.

In order to reduce the number of connections that are established, many MPI libraries use

Table 3.1: Percentage of process pairs that communicate in the NPB

Benchmark	CLASS	NPROCS	Percentage
Block Tridiagonal (BT)	D	256	5.5%
Embarrassingly Parallel (EP)	D	256	0.78%
Integer Sort (IS)	C	256	100%
Lower Upper (LU)	D	256	4.6%
Multi Grid (MG)	D	256	5.4%
Scalar Pentadiagonal (SP)	D	256	5.5%

lazy connect strategies. MPICH [23,45] and MPICH2 [47] fall into this category. By establishing connections only on demand, lazy connection establishment addresses scalability issues in an application-aware manner. It works especially well when each process only communicates with a few other processes, which is the case for many scientific applications. As shown in Table 3.1, fewer than 6% of process pairs actually communicate in many of the benchmarks included in the NAS Parallel Benchmarks (NPB) [13,75].

However, some applications generate all-to-all or more or less uniform communication patterns, and those applications will end up establishing a large number of connections even when connections are established only on demand. Some examples of such applications are n-body simulations, Integer Sort (IS) in the NPB and load balancing based on random work stealing [6]. Moreover, simple lazy connect strategies will not extend to wide-area environments, because lazy connects may fail due to firewalls or NAT. My proposed connection management scheme (Section 4.3) also uses a lazy connect strategy, but overcomes these problems by predetermining a small number of connectable candidate connections during initialization.

3.6 Rank assignment techniques

The assignment of ranks to processes has a large effect on the performance of MPI applications, for the following reasons:

- Many applications have non-uniform communication patterns, so some rank pairs communicate more than others.

- Some process pairs have higher communication costs than others. In a multi-cluster environment, communication between clusters is more expensive than communication within a cluster. Even in a single cluster environment, communication that spans multiple switches is more expensive than communication that is contained within a single switch.

A traditional rank assignment scheme is to sort processes by host name or IP address and to assign ranks in that order. There are two assumptions under this scheme:

1. Most communication takes place between processes with close ranks.
2. Processes with close host names or IP addresses have low communication cost.

Application programmers often take care to satisfy assumption 1, but they do not always do so nor should they have to. For example, in the NPB, Lower Upper (LU) satisfies assumption 1 but Block Tridiagonal (BT) does not (graphical representations of the benchmarks used in this dissertation are given in Appendix A.2). As for assumption 2, it may hold in a small Local Area Network (LAN), but host names and IP addresses have a much smaller correlation with communication cost in a Wide Area Network (WAN). Even within a LAN, virtualization techniques such as Virtual LAN (VLAN) [77] disrelate IP addresses from communication costs.

Hatazaki et al. [26] and Traff [76] have proposed topology-aware rank assignment schemes in the context of the topology creation functions of MPI (i.e., `MPI_Cart_create` and `MPI_Graph_create`). They formulate the rank assignment problem as a graph partitioning problem and use a variant of the Kernighan-Lin heuristic [36] to produce good solutions quickly. Unlike my proposed rank assignment scheme (Section 4.4), these schemes require communication costs and traffic patterns to be supplied manually; communication costs are supplied to the MPI library as a weighted graph, and traffic patterns are supplied via the topology creation function.

Bhanot et al. [5] have proposed a topology-aware rank assignment scheme for the Blue Gene/L supercomputer (BG/L) [4], which like the two previous schemes, formulates the rank assignment problem as a graph partitioning problem. The graph partitioning problem is then handled by a library called METIS [35, 49]. Like my proposed rank assignment scheme, this scheme can make use of traffic patterns from a profiling run. However, unlike my scheme, the network topology is limited to a torus and communication costs must be supplied manually.

Adaptive MPI (AMPI) [30, 31] is an adaptive message passing system that uses virtual processors. It performs load balancing by migrating virtual processors in a way that balances the execution times of physical processors while minimizing inter-processor communication. This load balancing scheme and my rank assignment scheme are similar in that they both adapt to applications by tracking the amount of communication that is performed. They differ, however, in the way that they treat communication costs. AMPI is designed for single cluster environments, so it assumes that the communication cost of every processor pair is the same. My rank assignment scheme, which is designed for multiple clusters, recognizes that different process pairs have different communication costs.

Chapter 4

Design and Implementation of MC-MPI

4.1 Overview

In this chapter, I present the design and implementation of Multi-Cluster MPI (MC-MPI). MC-MPI is a wide-area message passing library that implements most of the Message Passing Interface (MPI) [42] standard. The following are its two main features:

1. **Locality-aware connection management**

   MC-MPI uses a locality-aware connection management scheme to establish connections between processes. This scheme overcomes firewalls and Network Address Translation (NAT) [66] by constructing an overlay network, and achieves scalability by limiting the number of connections that each process establishes to $O(\log n)$ when the total number of processes is n. In order to achieve high performance with a limited number of connections, the connections that are established are selected in a locality-aware manner, based on latency and traffic information obtained from a short profiling run.

2. **Locality-aware rank assignment**

   MC-MPI uses a locality-aware rank assignment scheme to find a low-overhead mapping between ranks (process IDs) and processes. This scheme formulates the rank assignment problem as a Quadratic Assignment Problem (QAP) [38], and adapts to environments and to applications by setting up QAPs using the latency and traffic information obtained from the profiling run.

The latency and traffic information used by the proposed connection management and rank assignment schemes are obtained in the form of matrices. As illustrated in Figure 4.1, a short

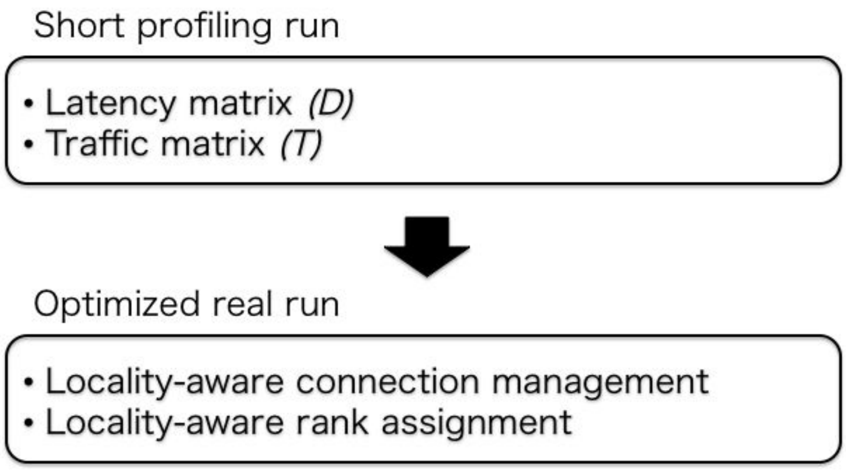


Figure 4.1: Profiling run and real run in MC-MPI

profiling run obtains a latency matrix and a traffic matrix, and those matrices are used to execute an optimized real run.

In addition to latency and traffic information, the proposed connection management scheme assumes that each process knows the endpoint of every other process (whether or not each endpoint is connectable need not be known—just a list of IP addresses and port numbers is sufficient). In order to provide processes a method for exchanging endpoints, MC-MPI uses a Grid shell called GXP [72] for job submission. GXP allows a user to perform Secure Shell (SSH) [80] logins to multiple nodes in multiple clusters. This creates a login tree that processes can use as a side channel to exchange endpoints at startup. Other job submission methods can be used as long as processes are brought up with some initial method to exchange their endpoints (e.g., a connection to the master process).

The rest of this chapter is organized as follows. First, in Section 4.2, I describe the latency and traffic matrices that are obtained from the profiling run. Then, in Section 4.3, I explain my connection management scheme. Finally, in Section 4.4, I explain my rank assignment scheme.

4.2 Profiling run

4.2.1 Latency matrix

Latency matrix D represents the latency between processes in the target environment. Each element d_{ij} represents the round-trip time (RTT) between processes i and j ($i, j = 0, 1, 2, \ldots, n-1$). Unfortunately, measuring the RTT of all process pairs takes a long time. Measuring the RTT

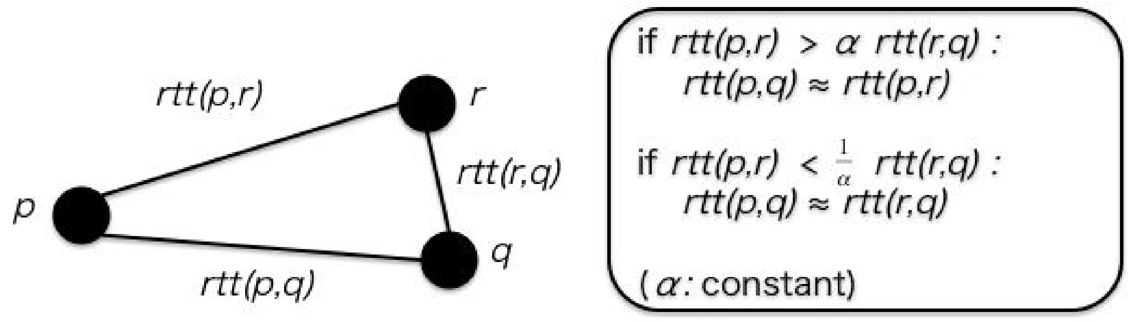


Figure 4.2: RTT estimation scheme

between faraway processes takes particularly a long time for the following reasons:

- The RTT between faraway processes is long, and measurement time is proportional to the RTT.
- Connections blocked by firewalls cause processes to wait for timeouts, and communication between faraway processes is more likely to be blocked than those between nearby processes.

Thus, in the proposed method, RTTs between faraway processes are estimated based on other RTTs (Figure 4.2). The algorithm that each process follows in order to measure and estimate RTTs is shown in Algorithm 1 (each process autonomously executes obtain_rtts (line 4)).

This estimation algorithm is based on the fact that the triangular inequality

$$|\mathit{rtt}(p,r) - \mathit{rtt}(r,q)| < \mathit{rtt}(p,q) < \mathit{rtt}(p,r) + \mathit{rtt}(r,q)$$

often holds for the RTTs of three processes p, q and r. In particular, when process p performs RTT measurements with another process r, p receives from r the set of $\mathit{rtt}(r,q)$ such that $\mathit{rtt}(p,q)$ is unknown to p but $\mathit{rtt}(r,q)$ is known to r (line 9). It then estimates $\mathit{rtt}(p,q)$ using $\mathit{rtt}(p,r)$ (measured) and $\mathit{rtt}(r,q)$ (received) as follows:

- If $\mathit{rtt}(p,r) > \alpha \mathit{rtt}(r,q)$, use $\mathit{rtt}(p,r)$ as an estimate for $\mathit{rtt}(p,q)$.
- If $\mathit{rtt}(p,r) < \frac{1}{\alpha}\mathit{rtt}(r,q)$, use $\mathit{rtt}(r,q)$ as an estimate for $\mathit{rtt}(p,q)$.
- Otherwise, $\mathit{rtt}(p,q)$ cannot be estimated accurately from $\mathit{rtt}(p,r)$ and $\mathit{rtt}(r,q)$.

Algorithm 1 Algorithm followed by each process p for RTT measurement and estimation

```
 1: // $rtt(p,r)$ : the RTT between $p$ and $r$
 2: // $S(p)$ : $\{q \mid rtt(p,q)$ is unknown to $p\}$
 3:
 4: procedure obtain_rtts():
 5:     $S(p) = \{$ all endpoints $\}$
 6:     while $S(p) \neq \phi$ do
 7:         select $r$ randomly from $S(p)$
 8:         $rtt(p,r) =$ measure_rtt($r$)
 9:         $T =$ receive_rtts($r$)
10:         for all $(q, rtt(r,q)) \in T$ do
11:             if $rtt(p,r) > \alpha rtt(r,q)$ then
12:                 $rtt(p,q) = rtt(p,r)$
13:                 $S(p) = S(p) - \{q\}$
14:             else if $rtt(p,r) < \frac{1}{\alpha} rtt(r,q)$ then
15:                 $rtt(p,q) = rtt(r,q)$
16:                 $S(p) = S(p) - \{q\}$
17:             end if
18:         end for
19:         $S(p) = S(p) - \{r\}$
20:     end while
21: endprocedure
22:
23: procedure measure_rtt($r$):
24:     measure $rtt(p,r)$ and return it
25: endprocedure
26:
27: procedure receive_rtts($r$):
28:     receive $\{(q, rtt(r,q)) \mid q \in S(p) \land q \notin S(r)\}$ from $r$ and return it
29: endprocedure
```

Here, α is a parameter that can be changed to improve the accuracy of RTT estimates at the cost of more measurements. The effects of changing α is discussed in Section 5.4, but a fixed value of $\alpha = 5$ is used elsewhere.

4.2.2 Traffic matrix

Traffic matrix T represents the traffic between processes in the target application. Each element t_{ij} represents the number of bytes sent from rank i to rank j during a trial run of the application ($i, j = 0, 1, 2, \ldots, n-1$). If the application is iterative, the trial run consists of one iteration of the main loop. If the application is not iterative, the trial run is a timed execution of the application. For example, in the experiments in Chapter 5, a one-iteration trial run is used for the NAS Parallel Benchmarks (NPB) [13, 75], while a five-second trial run is used for Distributed Verification Environment (DiVinE) [2, 28].

The idea behind obtaining T from a trial run is that many applications repeat similar communication patterns throughout their lifetime. Of course, the communication patterns of some applications are unpredictable. In that case, my connection management scheme performs optimizations using just D. Meanwhile, my rank assignment scheme only works when both D and T are available.

4.3 Locality-aware Connection Management

4.3.1 Basic workings

The proposed connection management scheme overcomes firewalls and NAT by constructing an overlay network, and achieves high scalability by only using $O(\log n)$ connections per process to construct the overlay network. It achieves locality-awareness and adaptivity by selecting the $O(\log n)$ connections based on latency matrix D and traffic matrix T.

In addition to only using $O(\log n)$ connections per process to construct the overlay network, my connection management scheme uses a lazy connect strategy in order to reduce the number of connections. In this strategy, the $O(\log n)$ connections are preselected during the initialization phase and are only established on demand. This causes few connections to be established when few process pairs communicate, while the number of connections is bounded even when many or all process pairs communicate.

The overall initialization procedure (i.e., `MPI_Init`) consists of the following three steps:

1. **Bounding graph construction** (Subsection 4.3.2)

   Each process selects a subset of other processes based on D and T, and attempts to establish temporary connections to them. The set of successful temporary connections is called the *bounding graph*.

2. **Routing table construction** (Subsection 4.3.3)

   Processes compute routes between all pairs of processes using edges of the bounding graph.

3. **Spanning tree construction** (Subsection 4.3.4)

   A *spanning tree* is created from the bounding graph. This spanning tree will act as a side channel to assist lazy connection establishment. All of the temporary connections except for those of a the spanning tree are closed.

The application starts with only the connections of the spanning tree established, and lazily establishes real connections between neighboring processes of the bounding graph (Subsection 4.3.5). Connections are never established between processes not neighboring in the bounding graph.

In the rest of this chapter, the terms *temporary connection* and *real connection* are used as follows:

Temporary connection: A transient connection established during bounding graph construction. It is used to construct the routing table, but it is not used to actually route application traffic.

Real connection: A permanent connection established on demand during execution of the application body. It is used to route application traffic.

4.3.2 Bounding graph construction

In order to construct the bounding graph, each process p selects a subset of other processes as shown below and attempts to establish temporary connections to them:

- Select all of the following $\beta - 1$ processes:

$$q_1, q_2, q_3, \ldots, q_{\beta-1}$$

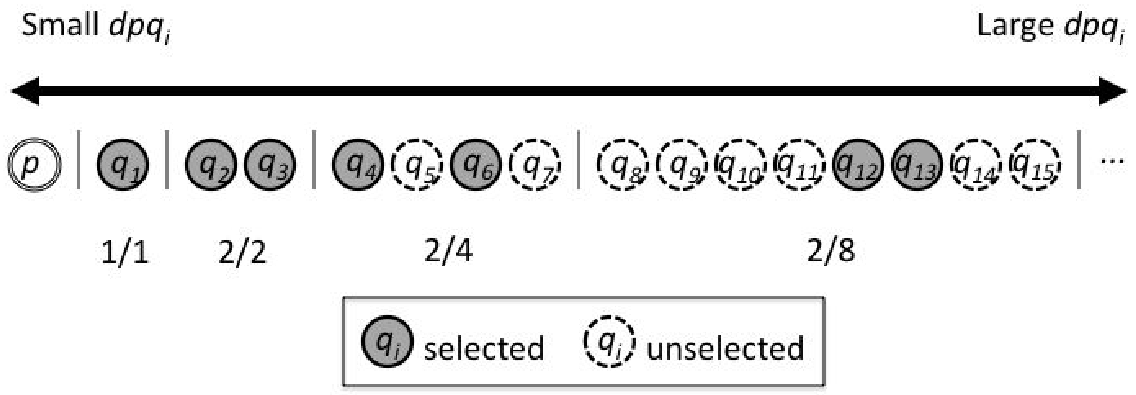

Small dpq_i
Large dpq_i
p
q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9 q_{10} q_{11} q_{12} q_{13} q_{14} q_{15} ...
1/1
2/2
2/4
2/8
q_i selected
q_i unselected

Figure 4.3: Process selection with $\beta = 2$

- Select β out of the following $2^{j-1}\beta$ processes ($j = 1, 2, 3, \ldots, \log_2 \frac{n}{\beta}$):

$$q_{2^{j-1}\beta}, q_{2^{j-1}\beta+1}, q_{2^{j-1}\beta+2}, \ldots, q_{2^j\beta-1}$$

Here, n is the total number of processes and β is a parameter that controls connection density. Moreover, $q_1, q_2, q_3, \ldots, q_i, \ldots, q_{n-1}$ are the $n-1$ processes besides p sorted in increasing order of d_{pq_i}. Figure 4.3 depicts process selection with $\beta = 2$.

By creating larger groups for processes with higher latency but selecting the same number of processes from each group, nearby processes are connected densely while faraway processes are connected sparsely. This results in routes with lower latency than if the same number of connections were selected randomly, and consumes less wide-area resources.

For applications that have unpredictable communication patterns, β processes are selected randomly from each group of $2^{j-1}\beta$. Meanwhile, for applications that have predictable communication patterns (e.g, iterative applications), T is used to achieve application awareness in addition to the environment awareness already achieved by using D. In this case, the probability that q_k is selected is proportional to t_{pq_k} ($k = 2^{j-1}\beta, 2^{j-1}\beta + 1, 2^{j-1}\beta + 2, \ldots, 2^j\beta - 1$).

In order to determine an upper bound on the number of connections that are established, assume that no connections are blocked by firewalls or NAT and that all attempts to establish temporary connections succeed. Also assume that the n processes are distributed equally among c clusters and that n is a power of 2. Then, as described below, $O(\log n)$ connections are established by each process, and $O(n \log c)$ inter-cluster connections are established by all processes collectively:

- Each process establishes $\beta \log_2 \frac{n}{\beta} + \beta - 1$ connections.

- Of those connections, $\beta\log_2 c$ of them are inter-cluster connections. Thus, a total of $\beta n\log_2 c$ inter-cluster connections are established by all processes collectively.

Fewer connections are established when communication between some processes is blocked by firewalls or NAT. Moreover, depending on the value of β and the number and location of firewalls, there is a probability that the bounding graph is disconnected. However, as discussed in Section 5.5, the disconnect probability of bounding graphs is low enough for practical use.

4.3.3 Routing table construction

Once the bounding graph has been constructed, routes using edges of the bounding graph are computed. Processes exchange link state messages, and every process computes the shortest path to every other process using Dijkstra's algorithm [14] with RTT as the metric. In finding routes, the fact that Transmission Control Protocol (TCP) [55] connections are bidirectional is leveraged; each connect has a direction, but once a connection has been established, it can be used in both directions. Thus, the bounding graph can be regarded as an undirected graph when computing routes. However, the direction in which each temporary connection was established is remembered, in order to assist lazy connection establishment (in order to prevent lazy connects from failing, real connections are established in the same direction as temporary connections).

4.3.4 Spanning tree construction

After the routing table has been constructed, all of the temporary connections except for those of a spanning tree are closed. The spanning tree, which processes will use as a side channel to assist lazy connection establishment, is created from the bounding graph as follows:

- Choose one process at random. This process is the root of the spanning tree.
- Find the set of edges included in the routes from the root process to all other processes. This set of edges is the spanning tree.

Creation of the spanning tree ends the initialization procedure.

4.3.5 Lazy connection establishment and message forwarding

Real connections, which are used for routing application traffic, are established on demand. The algorithm for forwarding messages while establishing connections on demand is

Algorithm 2 Algorithm for forwarding messages while establishing connections on demand

```
// BG : bounding graph
// ST : spanning tree
// req : message for requesting a connection
// self : the process executing a particular procedure

procedure forward_app_msg(src, dst, msg):
    next = get_next_hop(BG, dst)
    if not connected(next) then
        if connectable(next) then
            establish a real connection with next
        else
            next2 = get_next_hop(ST, next)
            send(ST, self, next, next2, req)
            accept a real connection from next
        end if
    end if
    send(BG, src, dst, next, msg)
endprocedure

procedure get_next_hop(G, dst):
    return the next hop to dst in G
endprocedure

procedure connected(next):
    if the real connection with next is already established then
        return TRUE
    else
        return FALSE
    end if
endprocedure

procedure connectable(next):
    if self established the temporary connection with next then
        return TRUE
    else
        return FALSE
    end if
endprocedure
```

Algorithm 2 (cont'd)

```
39: procedure send($G$, $src$, $dst$, $next$, $msg$):
40:    forward msg using the real connection with next
41:    upon receiving msg, next executes handle_msg($G, src, dst, msg$)
42: endprocedure
43:
44: procedure handle_msg($G, src, dst, msg$):
45:    if $G == BG$ then
46:       if $dst == self$ then
47:          deliver msg to the application
48:       else
49:          forward_app_msg($src, dst, msg$)
50:       end if
51:    else
52:       if $dst == self$ then
53:          establish a real connection with src
54:       else
55:          $next2$ = get_next_hop($ST, dst$)
56:          send($ST, src, dst, next2, req$)
57:       end if
58:    end if
59: endprocedure
```

shown in Algorithm 2. When the application sends a message *msg* from p to q, p executes forward_app_message(p, q, msg) (line 6). After that, each process along the route from p to q executes forward_app_message (line 49), and *msg* is recursively delivered to q. The process executing forward_app_message can immediately forward *msg* if the real connection with the next hop *next* is already established (line 17). If the real connection with *next* is not established yet, it is established before *msg* is forwarded (lines 9–14).

If the temporary connection between *self* and *next* was established by *self*, the real connection between *self* and *next* can also be established by *self* (line 10). However, if the temporary connection was established by *next*, it is not known whether a connection can be established from *self* to *next*. Thus, *self* sends a message to *next* using the spanning tree, and has *next* establish the real connection (lines 12–14).

Real connections are established along all edges of the bounding graph when many process pairs communicate, but they are only established along the edges that are actually used when only a few process pairs communicate. Even when only a few process pairs communicate, temporary connections are established along all edges of the bounding graph. However, temporary connections do not require as much memory as real connections, because they do not require large TCP buffers like real connections.

Moreover, when forwarding small messages, the entire message is received before it is forwarded (line 49). Meanwhile, large messages are split up into small segments and pipelined in order to improve throughput.

4.4 Locality-aware Rank Assignment

4.4.1 Basic workings

As discussed in Chapter 2, rank assignment schemes based on host names or IP addresses do not work well, because their assumptions about communication costs and traffic patterns are too simplistic. Meanwhile, there are also topology-aware methods that do not make these simplistic assumptions [5, 26, 76], but these methods require information such as communication costs to be supplied manually.

Unlike existing topology-aware methods, the proposed rank assignment scheme requires no information to be supplied manually. It uses latency matrix D and traffic matrix T, obtained from the profiling run, to set up a QAP that reflects the application and its execution environment (Subsection 4.4.2). Then a good approximate solution to the QAP is found in a short amount of time using heuristics (Subsection 4.4.3)

4.4.2 Setting up QAPs for rank assignment

In the proposed rank assignment scheme, the communication overhead of a process pair is defined as the product of the expected traffic and the communication cost, and the communication overhead of the entire application is defined as the sum of the overheads of all process pairs. Here, the expected traffic between ranks i and j and the communication cost of processes i and j are defined as follows:

Expected traffic: t_{ij} (the number of bytes sent from rank i to rank j during the profiling run)

Communication cost: d_{ij} (the RTT between processes i and j measured or estimated during the profiling run)

The problem of finding the rank-process mapping with the lowest communication overhead then becomes a matter of finding a permutation p of the set $N = \{0, 1, 2, \dots, n-1\}$ that minimizes

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} t_{ij} d_{p(i)p(j)}$$

This is a problem in combinatorial optimization known as the Quadratic Assignment Problem (QAP), and was first introduced by Koopmans and Beckman [38].

The rationale behind using RTT for communication cost is as follows. The RTT between two processes is related to the number of communication links that are in between the two processes, and the more links there are, the more likely that traffic between the two processes will coincide with that of other processes. Thus, by assigning rank pairs with high traffic volume to process pairs with short RTTs, my method optimizes for bandwidth as well as latency. Point-to-point bandwidth is another possible metric for communication cost, but it does not work as well as RTT, because it does not account for what happens when multiple process pairs share the same link.

4.4.3 Solving QAPs

Like many other problems in combinatorial optimization, the QAP is non-deterministic polynomial time hard (NP-hard), and it is unfeasible to find the optimal solution when n is large (problems of $n > 20$ are not practically solvable). However, many good heuristics for finding good suboptimal solutions have been proposed. Some such heuristics include cutting plane algorithms [3, 8], simulated annealing algorithms [10, 43] and genetic algorithms [44, 79].

In my rank assignment scheme, a library developed by Resende and Pardalos [59] is used. This library uses a heuristic called Greedy Randomized Adaptive Search Procedure (GRASP) to obtain approximate solutions to QAPs. Tested against QAPLIB [9, 56], a publicly available collection of QAPs, it was able to obtain approximate solutions that were within 3 percent of the best known solution for instances of up to $n = 256$ in under 5 seconds. A problem size of n means that there are n cores available, so n cores were used for computing QAP solutions (GRASP can be parallelized easily). Appendix A.1 describes the results in more detail.

Chapter 5

Performance Evaluation of MC-MPI

5.1 Overview

In this chapter, I present experimental results concerning Multi-Cluster MPI (MC-MPI). First, in Section 5.2, I describe the wide-area environment used in my experiments. Then, in Section 5.3, I describe the NAS Parallel Benchmarks (NPB) [13, 75] and Distributed Verification Environment (DiVinE) [2, 28], the benchmarks used in my experiments. In Sections 5.4 and 5.5, I evaluate my RTT estimation scheme and the disconnect probability of bounding graphs. Finally, in Sections 5.6, 5.7 and 5.8, I evaluate my connection management and rank assignment schemes using the NPB and DiVinE.

5.2 Experimental environment

The experimental environment consisted of 256 cores distributed equally across 4 clusters. Table 5.1 shows the specifications of each cluster, and Figure 5.1 shows the round-trip time (RTT) and the bandwidth of each cluster and between each pair of clusters. In terms of latency, there was an order of magnitude difference between the RTT between clusters (0.3 to 6.4 millisec-

Table 5.1: Specifications of each cluster

Cluster	Nodes (Cores)	CPU	RAM	OS (TCP)	Network
Chiba	32 (64)	Core 2 Duo (2.13GHz)	4GB	Linux 2.6.18 (BIC)	GbE
Istbs	64 (64)	Xeon MP (2.4GHz)	2GB	Linux 2.6.18 (BIC)	GbE
Kototoi	16 (64)	Xeon 5140 (2.33GHz)	8GB	Linux 2.6.18 (BIC)	GbE
Suzuk	32 (64)	Core 2 Duo (2.13 GHz)	4GB	Linux 2.6.18 (BIC)	GbE

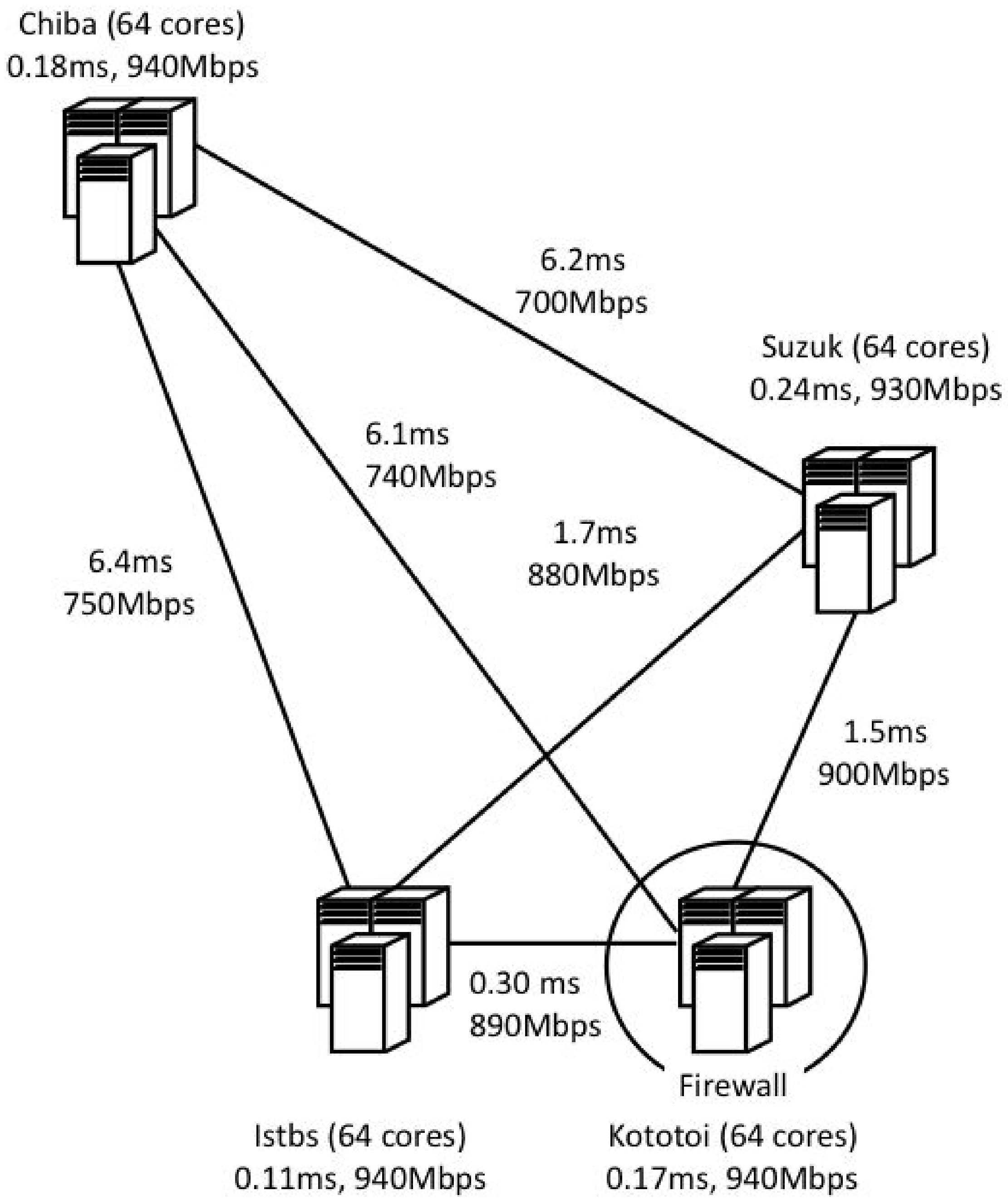


Figure 5.1: RTT and bandwidth of each cluster and between each pair of clusters

onds) and that within a cluster (0.11 to 0.24 milliseconds). In terms of bandwidth, the point-to-point bandwidth between clusters (700 to 900 Mbps) was only slightly lower than that within a cluster (930 to 940 Mbps). However, multiple communicating process pairs always had to share bandwidth between clusters but they often did not have to within a cluster. Moreover, there was a firewall around Kototoi, and that firewall permitted outgoing connections but denied incoming connections.

5.3 Description of benchmarks

5.3.1 NPB

The NAS Parallel Benchmarks (NPB) are a set of benchmarks derived from important classes of aerophysics applications [13, 75]. It contains both kernel benchmarks and simulated computational fluid dynamics (CFD) applications. Below is a brief description of the benchmarks used in the experiments in this chapter:

Block Tridiagonal (BT): BT is a simulated CFD application that solves three sets of uncoupled systems of equations. Its structure is similar to that of SP, except that the systems of equations are block tridiagonal instead of scalar pentadiagonal. It requires a square number of processes.

Embarrassingly Parallel (EP): EP is an "embarrassingly parallel" kernel. It provides an estimate of the upper achievable limits for floating point performance (i.e., the performance without significant inter-process communication). It runs on any number of processes.

Integer Sort (IS): IS is a kernel that performs a parallel sort of keys, in which the keys are initially distributed equally across processes. It requires a power-of-two number of processes.

Lower Upper (LU): LU is a simulated CFD application that performs a 2D partitioning of the grid onto processes and performs Successive Over-Relaxation (SOR). It requires a power-of-two number of processes.

Multi Grid (MG): MG is a simple 3D multigrid kernel that requires highly structured long distance communication and tests both short and long distance data communication. It requires a power-of-two number of processes.

Scalar Pentadiagonal (SP): SP is a simulated CFD application that solves three sets of uncoupled systems of equations. Its structure is similar to that of BT, except that the systems of equations are scalar pentadiagonal instead of block tridiagonal. It requires a square number of processes.

In my experiments, I used NPB3.2-MPI, the Message Passing Interface (MPI) [42] implementation of the NPB. In NPB3.2-MPI, BT, EP, LU, MG and SP are written mainly using point-to-point communication primitives. The four most used communication primitives in these benchmarks are `MPI_Irecv`, `MPI_Isend`, `MPI_Recv` and `MPI_Send`. Meanwhile, IS is written mainly using collective communication primitives. `MPI_Allreduce`, `MPI_Alltoall` and `MPI_Alltoallv` account for almost all of the communication in IS.

5.3.2 DiVinE

In order to show that my work is relevant to a wider range of applications than just those of the NPB, I also used Distributed Verification Environment (DiVinE) [2, 28] as a benchmark. DiVinE is a set of enumerative linear temporal logic (LTL) model checking tools for verification of concurrent systems.

DiVinE provides many tools, including one specifically tailored for shared memory architectures (DiVinE Multi-Core) and one specifically tailored for efficient usage of external memory devices (DiVinE I-O). I used DiVinE Cluster, the tool specifically tailored for distributed memory architectures. DiVinE Cluster is implemented using MPI, and is written mainly using point-to-point communication primitives. The four most used communication primitives are `MPI_Isend`, `MPI_Iprobe`, `MPI_Recv` and `MPI_Test`.

DiVinE Cluster implements multiple algorithms, including one based on negative cycles and one based on maximal accepting predecessors (MAP). Of these algorithms, I used One Way Catch Them Young (OWCTY), the algorithm based on strongly connected components. A look at the traffic matrix produced by the other algorithms suggest that they would produce similar results, but this has not been verified.

The model that is checked using OWCTY is that of an elevator controller. Given the number of floors, the number of people and the strategy for controlling the elevator, the following property is checked: if a person is waiting, he/she will be served eventually. The source code for this model was taken from BEnchmarks for Explicit Model checkers (BEEM) [27, 51], but the parameters were changed to those shown in Table 5.2 to increase the size of the problem.

Table 5.2: Parameters used for the elevator controller model

Floors	Persons	Strategy
10	3	0

BEEM provides many other models, including those for communication protocols and mutual exclusion algorithms, but the communication patterns that OWCTY produces for these models are similar to those that OWCTY produces or the elevator controller model.

5.4 RTT measurement time

Figure 5.2 shows the time required to acquire latency matrix D with various values of α. When α was large, many measurements were performed between faraway processes (i.e., processes in different clusters). Because of the long RTT of faraway processes, these measurements took a long time (e.g., the RTT between Chiba and Istbs was 36 times as long as the RTT within Istbs). Moreover, the firewall around Kototoi caused attempts to establish connections from other clusters to stall until they timed out, further increasing the required time. Meanwhile, when α was small, few measurements were performed between faraway processes, allowing D to be acquired in a short time.

In the following experiments, a fixed value of $\alpha = 5$ is used to reduce the number of measurements between faraway processes. With $\alpha = 5$, there is an estimation error of up to 25%. However, this is sufficient for the proposed connection management and rank assignment schemes, because it is accurate enough to distinguish between intra-cluster and inter-cluster RTTs.

5.5 Bounding graph disconnect probability

This section presents the results of a simulation that shows that the disconnect probability of bounding graphs is low enough for practical use. In this simulation, the disconnect rate was computed for the following two environments:

1FW: An environment that closely resembles the real environment used in the other experiments in this section. There are 4 clusters, there is a firewall around 1 of the clusters, and the RTTs between nodes are the same as those in Figure 5.1.

3FW: An environment that is basically the same as 1FW, but with more blocked communication. Instead of there being a firewall around 1 of the clusters, there is a firewall around 3 of the clusters (no firewall is placed around the 4th cluster, because that would prevent the clusters from being connected regardless of the method used).

The two parameters that were studied were β and the number of processes. The values of β that were simulated were 1, 2 and 4, and the number of processes that were simulated were 8, 16, 32, 64 and 128 (as discussed later, larger values of β and number of processes did not result in disconnected graphs).

For each pair of parameters, 10^8 bounding graphs were constructed, and the number of disconnected graphs were counted. With 1FW, no bounding graphs were disconnected regardless of the value of β or the number of processes. With 3FW, some bounding graphs were disconnected, as shown in Figure 5.3. Yet even with 3FW, increasing the number of processes or β resulted in fewer disconnected bounding graphs. With $\beta = 4$, no bounding graphs were disconnected regardless of the number of processes. Similarly, with 128 processes, no bounding graphs were disconnected regardless of the value of β.

5.6 Connection management performance

In this experiment, I evaluated my connection management scheme by measuring the performance of the NPB and DiVinE with varying numbers of connections. The following three methods were compared:

MC-MPI: My locality-aware connection management scheme, described in Section 4.3. The maximum percentage of connections allowed (the percentage of process pairs selected during bounding graph construction) was varied by using different values of β as shown in Table 5.3. Because of lazy connection establishment, fewer connections were actually established in most cases.

Random: A locality-unaware scheme in which the bounding graph was constructed by randomly selecting some percentage of connections. As in MC-MPI, these connections were established lazily.

Manual A scheme in which the processes that perform forwarding were manually configured. Processes connected all-to-all within clusters, but only the forwarding processes

Table 5.3: Percentage of connections selected with various values of β ($n = 256$)

β	Percentage
2	12.2%
4	20.1%
7	30.9%
10	39.1%
15	50.7%
21	60.6%
28	69.4%
38	79.7%
55	89.9%
128	100%

connected with processes in other clusters. The number of forwarding processes was varied from 1 to 60 per cluster, and each forwarding process connected with exactly one forwarding process in every other cluster. This method is representative of existing wide-area-enabled MPI libraries with forwarding mechanisms (e.g., GridMPI). Again, all connections were established lazily.

Figures 5.4 to 5.11 show the results. The graphs on the left side compare the performances of MC-MPI, Random and Manual. In these graphs, the horizontal axis is the maximum percentage of connections allowed, and the vertical axis is the performance relative to when all connections were allowed. Allowing all connections is representative of MPI libraries that require all-to-all connectivity (e.g., MPICH-G2). Meanwhile, the graphs on the right side show just the performance of Manual. In these graphs, the horizontal axis is again the maximum percentage connections allowed, but the vertical axis is the number of forwarding processes per cluster.

As a general trend, changing the number of forwarding processes drastically changed the performance of Manual, underscoring the difficulties of manual configuration. Having only a small number of forwarding processes resulted in lower performance for many of the benchmarks, because of the following reasons:

- The inter-cluster bandwidth could not be fully utilized with a small number of forwarding processes. One reason for this is that a forwarding process had to both receive and send each message, preventing it from handling as much data as a non-forwarding process could. Another reason is that when competing with the traffic of other applications, only a small percentage of the bandwidth could be claimed with a small number of streams.

- The forwarding processes became heavily loaded, slowing down any computation that those processes also had to handle.

Although increasing the number of forwarding processes increased performance, Manual was not able to perform better than MC-MPI even when all processes were designated as forwarding processes.

Manual particularly had problems with DiVinE (Figure 5.11). With 1 and 10 forwarding processes per cluster, the elevator controller model could not be checked completely, because the forwarding processes ran out of memory. This occurred because the forwarding processes received data from within their own clusters much more quickly than they could forward data outside their clusters. Meanwhile, MC-MPI and Random did not experience this problem, because there were many more than 10 forwarding processes per cluster, even when only 10% of the connections were allowed. MC-MPI and Random may also experience out-of-memory errors if the forwarding processes received data even more quickly, and the real solution to this problem is to implement flow control. However, existing MPI libraries typically do not implement flow control, because it introduces new problems such as deadlocks. Thus, I do not claim that MC-MPI solves this problem, but do claim that it fares better than simplistic methods such as Manual.

The general trend for Random was that its performance dropped when the number of connections was limited. Meanwhile, the performance of MC-MPI did not drop even when the number of connections was limited. For example, limiting the number of connections caused a sharp decrease in the performance of LU when Random was used, but not when MC-MPI was used (Figure 5.7a). This is because many messages had to be forwarded with Random, but all messages were delivered directly with MC-MPI. MC-MPI was able to deliver all messages directly even when only 10% of the connections were allowed, because each process only communicated with a small number of processes, and MC-MPI was able to select those processes based on T (LU is a benchmark that performs SOR, so each process performed most of its communication with just 4 processes). Similar results were obtained for BT, MG and SP (Figures 5.4a, 5.8a and 5.9a).

For IS, limiting the number of connections greatly improved the performance of both MC-MPI and Random (Figure 5.6a). This is because IS is a benchmark that performs a lot of collective communication. Figure 5.10 shows the results for when just the collective operations of IS were executed, and the results were similar to those of IS (the parameters of the collective operations are shown in Table 5.4). During collective communication, many processes

Table 5.4: Parameters of the collective operations used in IS

Operation	Data type	Send count (Number of bytes)
MPI_Allreduce	MPI_INT	1029 (4117B)
MPI_Alltoall	MPI_INT	1 (4B)
MPI_Alltoallv	MPI_CHAR	1024–3072 (1024–3072B)

communicate simultaneously, so the bandwidth available to each process is limited. However, the Transmission Control Protocol (TCP) [55] operates independently for each connection, so each connection may send too eagerly at the start of collective communication. Thus, congestion is worse when there is a large number of connections. While both MC-MPI and Random performed better with fewer connections, MC-MPI performed better than Random because it established fewer inter-cluster connections, where bandwidth was particularly limited.

When MC-MPI was used for DiVinE, limiting the percentage of connections to 50% increased performance, but further reduction decreased performance (Figure 5.11a). A small reduction increased performance because it helped reduce congestion, but a larger reduction decreased performance because the overhead of forwarded messages outweighed the benefits of reduced congestion. Yet with any magnitude of reduction, MC-MPI performed better than when all connections were allowed. Moreover, the curves of MC-MPI and Random have similar shapes, but MC-MPI consistently outperformed Random with the same number of connections.

Finally, limiting the number of connections had no impact on the performance of EP regardless of the method used, because it involved little communication (Figure 5.5).

5.7 Lazy connection establishment performance

While MC-MPI succeeded in delivering messages directly in BT, LU, MG and SP by selecting the processes that establish connections based on T, simply establishing connections on demand (i.e., the strategy of MPICH) would not have worked. First, the firewall around Kototoi would have caused some lazy connects to fail. Moreover, even in the absence of the firewall, a simple lazy connect strategy would have resulted in a large number of connections for applications with all-to-all or more or less uniform communication patterns.

In order to confirm that my lazy connect strategy works better than such a simple strategy, I compared the two following methods:

MC-MPI: My lazy connect strategy, which pre-determines a small number of candidate connections during initialization. $\beta = 7$ was used, limiting the number of connections to approximately 30%.

MPICH-like: The lazy connect strategy used by MPICH, which establishes any connection on demand.

The firewall around Kototoi was removed for this experiment, because MPICH-like requires all-to-all connectivity. Note that MC-MPI would have worked even with the firewall in place. In fact, in the environment of Figure 5.1, MC-MPI would have established the same exact set of connections with or without the firewall. The only difference is that with the firewall, MC-MPI would have established some of the connections in the reverse direction with the help of the spanning tree.

Figure 5.12 shows the results. For BT, EP, MG, LU and SP, MC-MPI and MPICH-like resulted in a similar number of connections and similar performance, showing that MC-MPI works just as well as MPICH-like for applications in which few process pairs communicate. For IS and DiVinE, in which all process pairs communicate, MPICH-like resulted in all process pairs establishing connections. Meanwhile, MC-MPI successfully limited the percentage of established connections to 30%. Moreover, by avoiding congestion, MC-MPI performed 1.83 times as well as MPICH-like for IS and 1.17 times as well as MPICH-like for DiVinE.

5.8 Rank assignment performance

In this experiment, I evaluated my rank assignment scheme by measuring the performance of the NPB with various rank assignments. The following three methods were compared:

Random: A rank assignment scheme that assigns ranks to processes randomly.

Hostname: The rank assignment scheme based on host names, described in Subsection 3.6. In addition to an assignment based on real host names, assignments based on virtual host names, in which host names were swapped on a per cluster basis, were tested. Of the ${}_4P_4 = 24$ sets of virtual host names, the best performing one is called Hostname (Best), and the worst performing one is called Hostname (Worst).

QAP (MC-MPI): My locality-aware rank assignment scheme, described in Section 4.4.

For LU, the rank assignments based on host names (Hostname, Hostname (Best) and Hostname (Worst)) performed 2.1 to 3 times better than Random (Figure 5.13d). This is because most of the communication took place between processes with close ranks (Figure A.4), and the assignments based on host names placed processes with close ranks in the same cluster. While all 24 assignments based on host names performed better than Random, some performed better than others, depending on whether close ranks were assigned to processes in close clusters. MC-MPI was able to perform almost as well as Hostname (Best). Similar results were obtained for MG (Figure 5.13e).

For BT, the rank assignments based on host names again performed better than Random (Figure 5.13a). This time, however, MC-MPI performed 20% better than even Hostname (Best). This is because a significant amount of communication of took place between processes with distant ranks (Figure A.1). For example, the ranks with which rank 0 communicated mainly were 1, 15, 16, 31, 240 and 241. Similar results were obtained for SP (Figure 5.13f).

For EP, the three rank assignment schemes resulted in the same performance (Figure 5.13b), because it involved little communication (Figure A.2). For IS, the three schemes resulted in the same performance (Figure 5.13c), because it had a uniform communication pattern (Figure A.3).

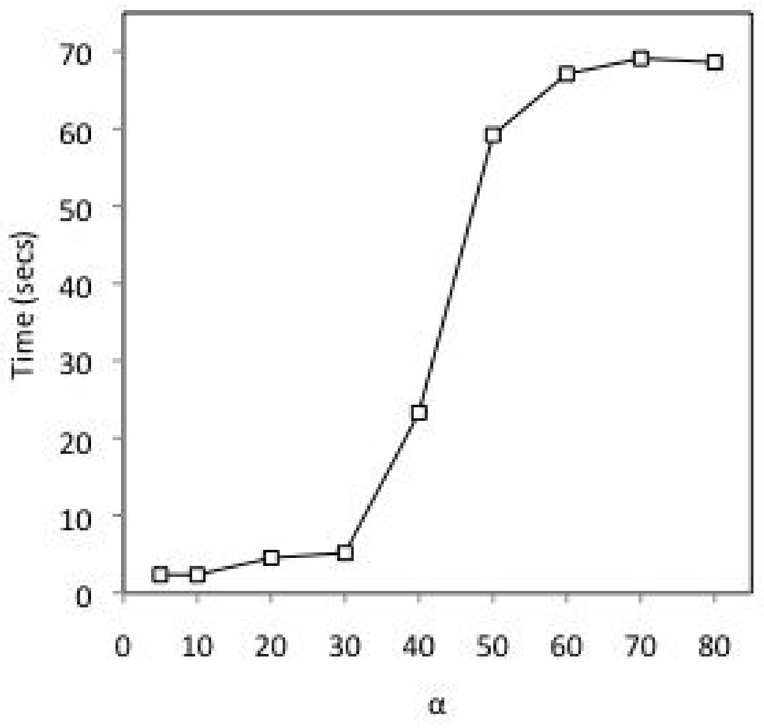


Figure 5.2: Time required to obtain D with various values of α

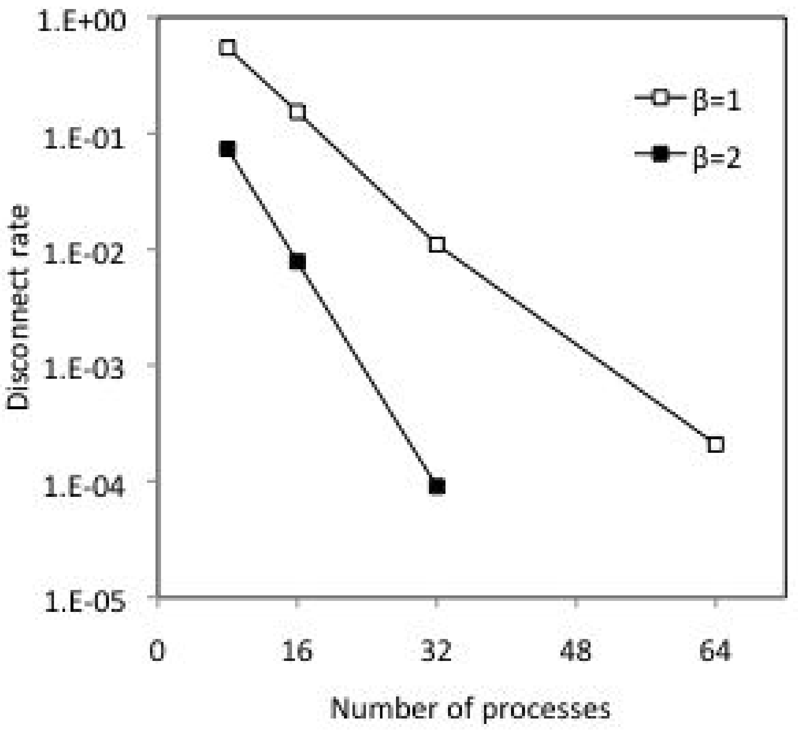
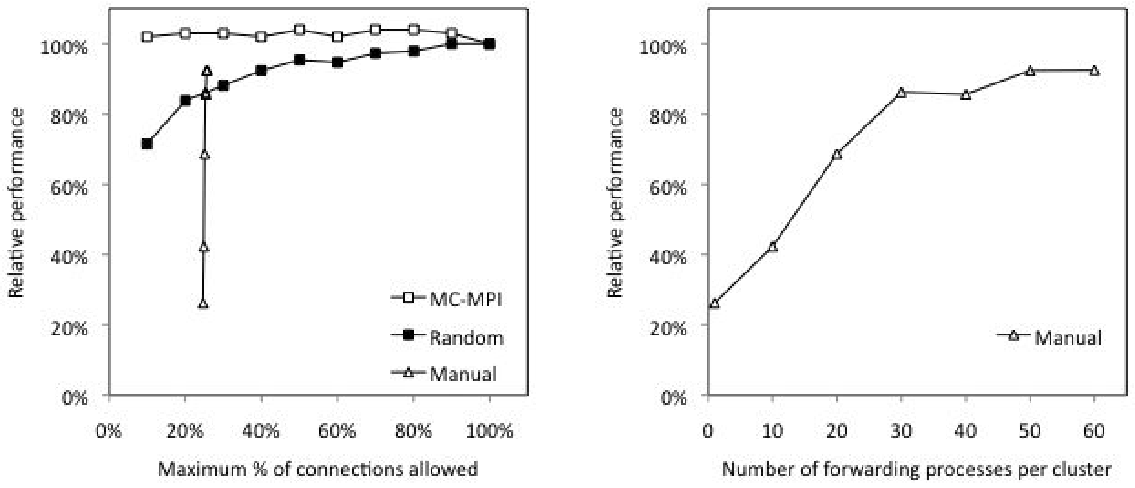
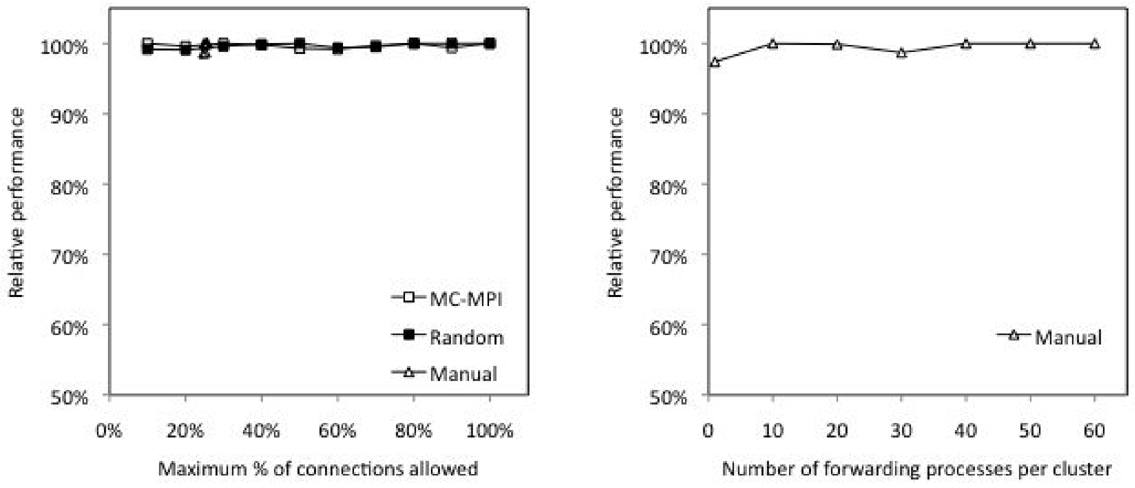


Figure 5.3: Disconnect rate of bounding graphs with 3FW



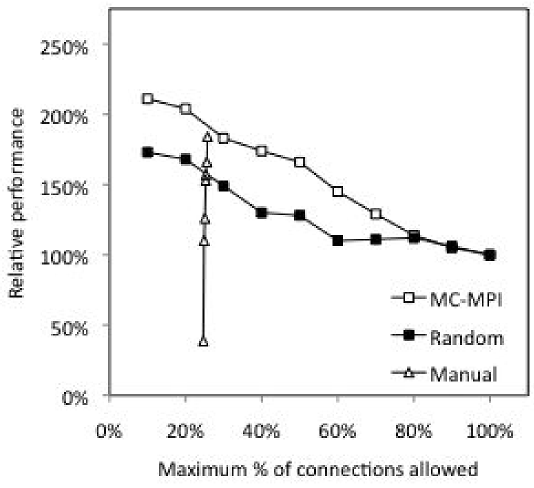
a. MC-MPI, Random and Manual b. Manual

Figure 5.4: Performance of BT (CLASS=D, NPROCS=256) with varying numbers of connections

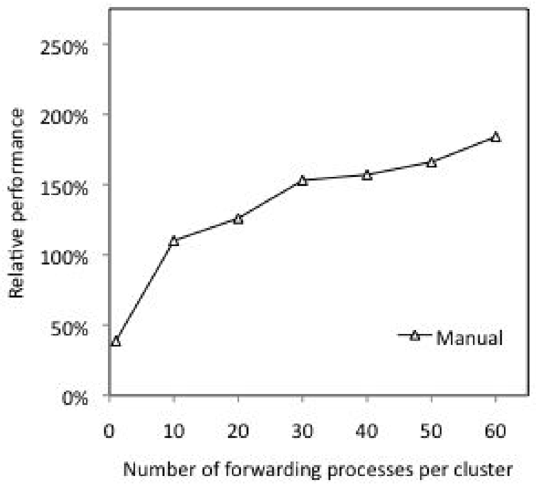


a. MC-MPI, Random and Manual b. Manual

Figure 5.5: Performance of EP (CLASS=D, NPROCS=256) with varying numbers of connections

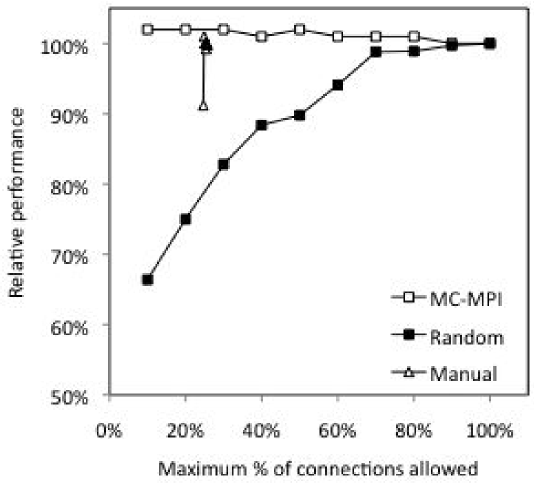


a. MC-MPI, Random and Manual

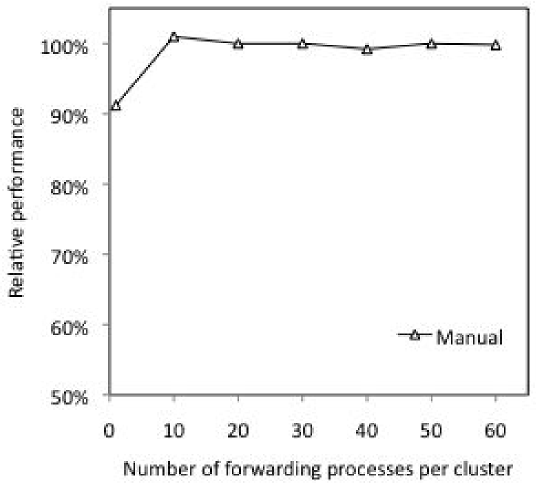


b. Manual

Figure 5.6: Performance of IS (CLASS=C, NPROCS=256) with varying numbers of connections

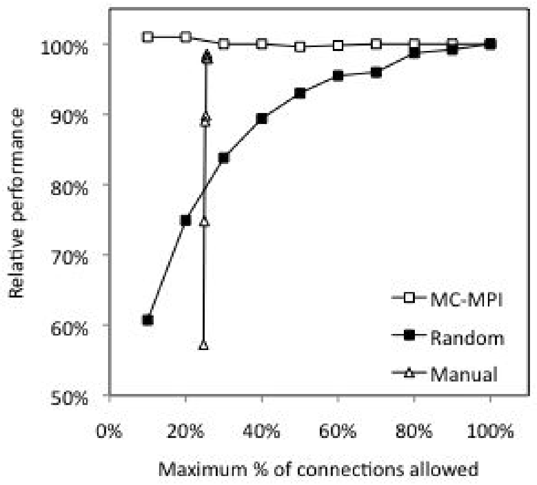


a. MC-MPI, Random and Manual

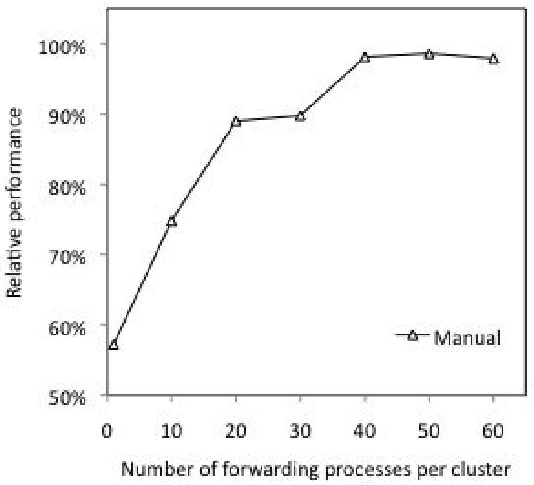


b. Manual

Figure 5.7: Performance of LU (CLASS=D, NPROCS=256) with varying numbers of connections

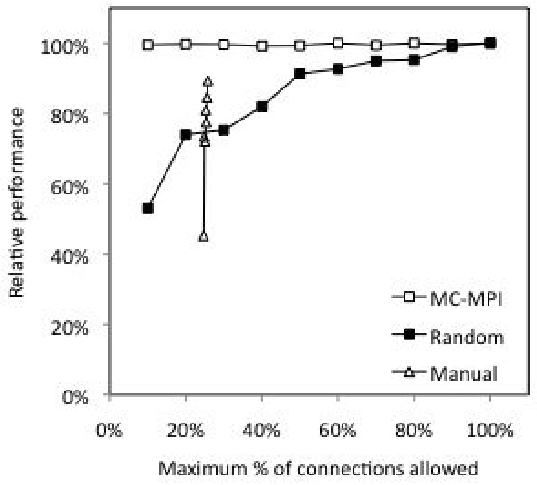


a. MC-MPI, Random and Manual

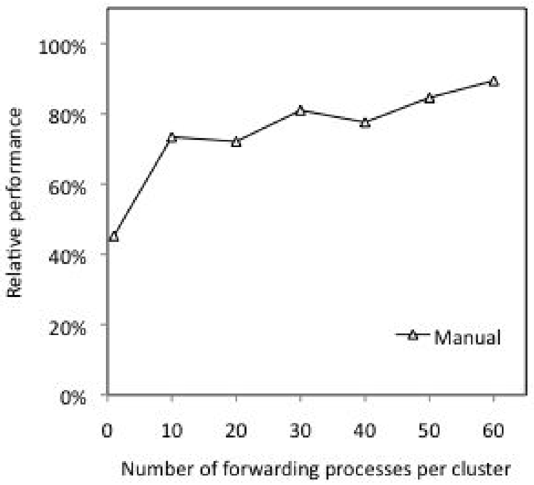


b. Manual

Figure 5.8: Performance of MG (CLASS=D, NPROCS=256) with varying numbers of connections

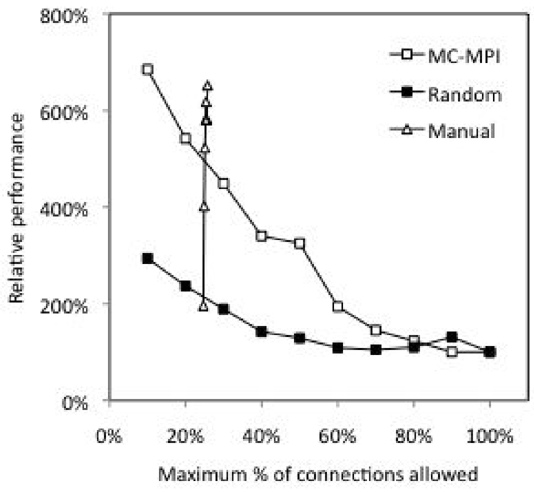


a. MC-MPI, Random and Manual

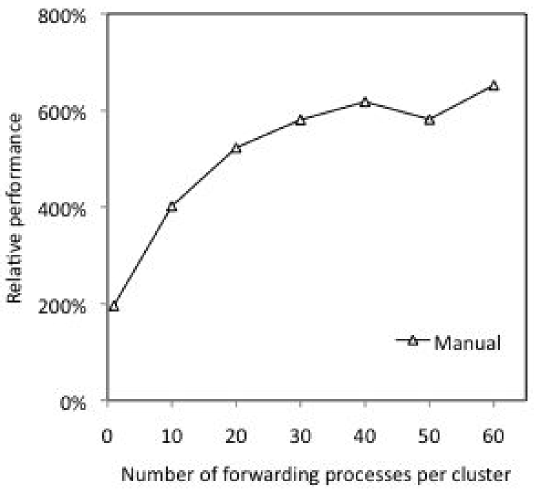


b. Manual

Figure 5.9: Performance of SP (CLASS=D, NPROCS=256) with varying numbers of connections

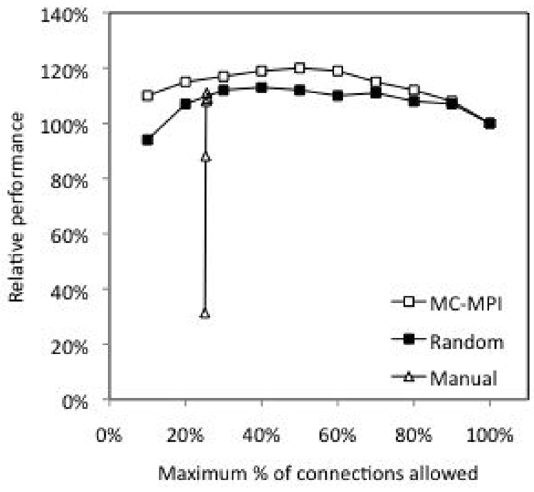


a. MC-MPI, Random and Manual

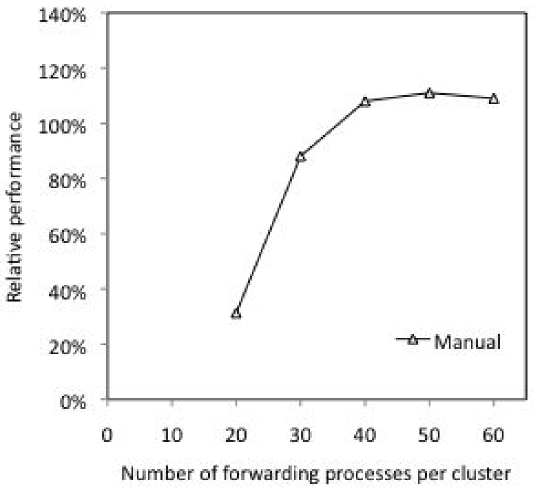


b. Manual

Figure 5.10: Performance of `MPI_Allreduce`-`MPI_Alltoall`-`Alltoallv` with varying numbers of connections

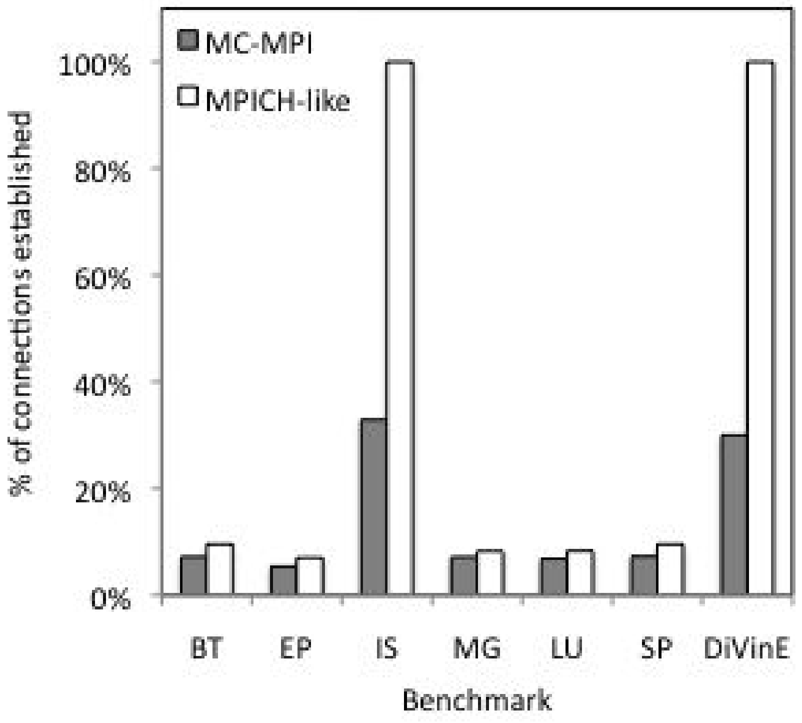


a. MC-MPI, Random and Manual

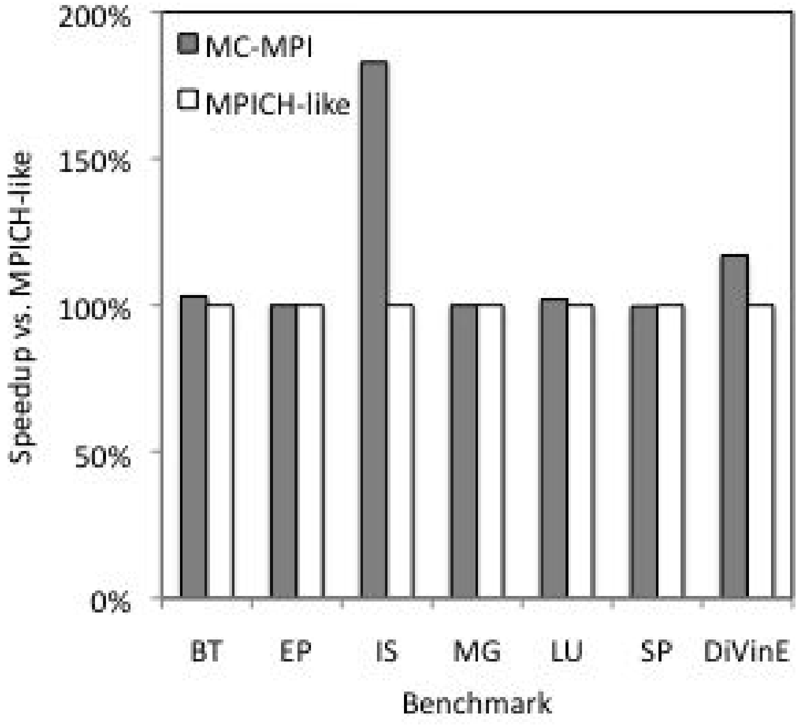


b. Manual

Figure 5.11: Performance of DiVinE with varying numbers of connections

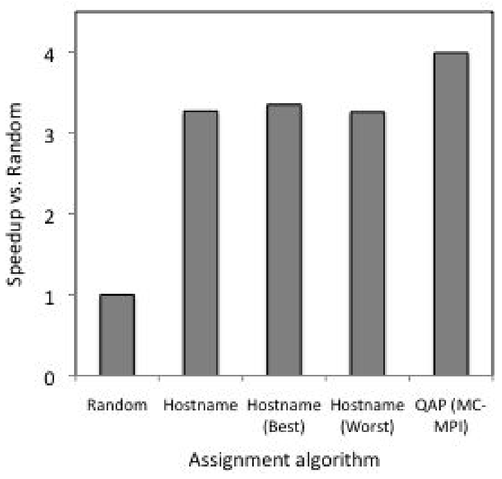


a. Percentage of connections established

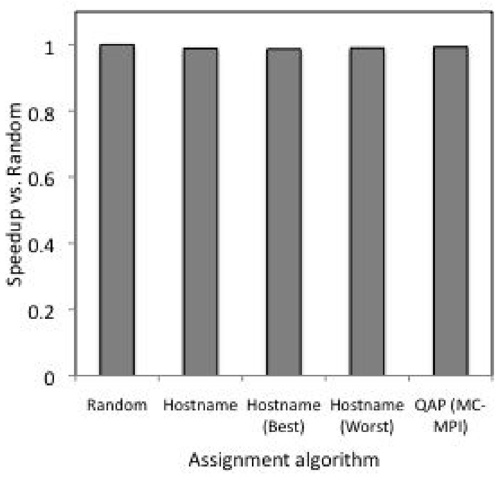


b. Speedup vs. MPICH-like

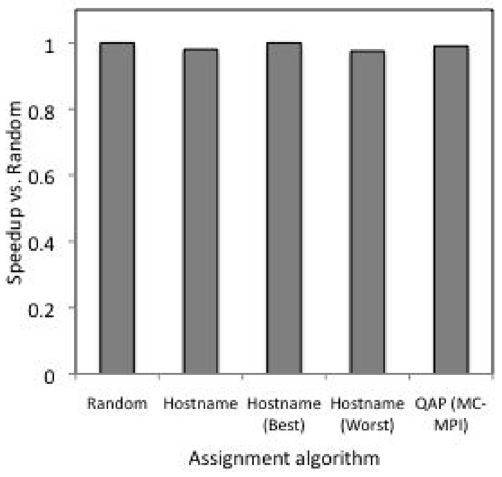
Figure 5.12: Comparison of lazy connection establishment methods



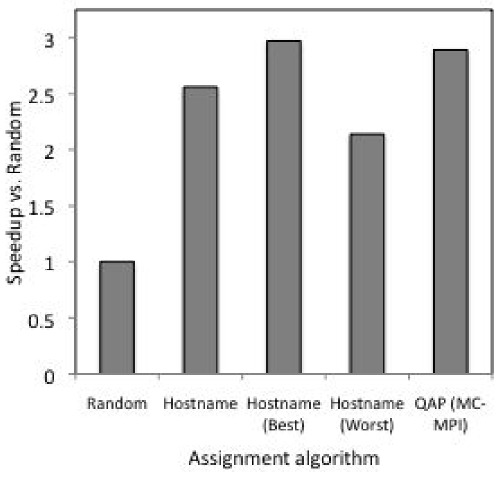
a. BT



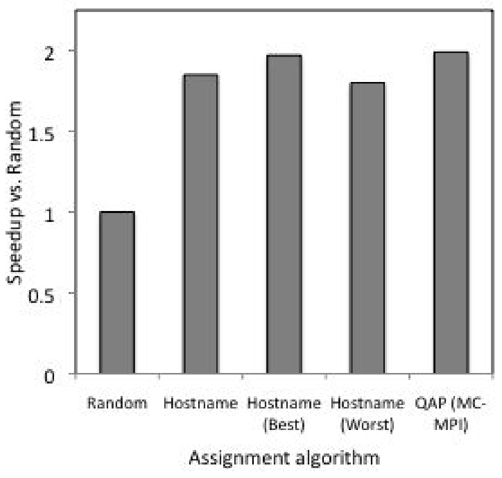
b. EP



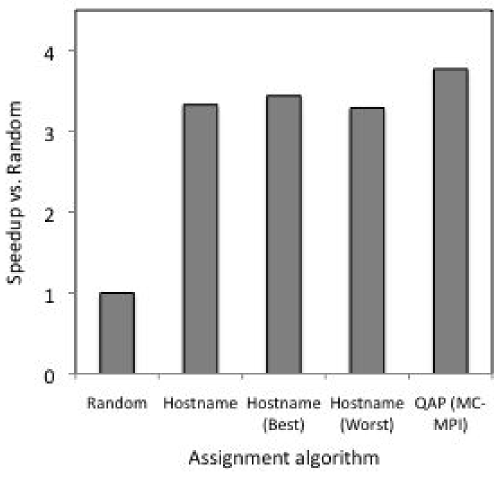
c. IS



d. LU



e. MG



f. SP

Figure 5.13: Performance of the NPB with various rank assignments

Chapter 6

Scalable Sockets

6.1 Design and implementation

6.1.1 Overview

While Multi-Cluster MPI (MC-MPI) solves many problems concerning connectivity, scalability, locality and adaptivity, those solutions can only be enjoyed by Message Passing Interface (MPI) [42] applications. Thus, in order to increase the number of applications that are supported, I have used my connection management scheme (Chapter 4.3) to implement a wide-area Sockets [53] library called Scalable Sockets (SSOCK).

Theoretically, a Sockets library with wide-area-enabled connection management can turn a local-area MPI library into one with wide-area-enabled connection management. Thus, ultimately, SSOCK would encompass many of the features of MC-MPI (of course, MPI-specific issues such as rank assignment and collective operations need to be handled separately). Unfortunately, SSOCK is currently only a prototype implementation and is missing some of the trivial functionalities of a full Sockets library, so simply linking SSOCK to MPICH [23, 45] or MPICH-2 [47] will not make it wide-area enabled. However, most of the important functionality of a Sockets library has already been implemented, allowing me to conduct experiments to evaluate its performance.

SSOCK implements most of the Sockets API, but also provides the following additional features:

- Allows any node to connect to any other node even in the presence of firewalls and Network Address Translation (NAT) [66].
- Allows all nodes to connect to each other without reaching various limits of the system.

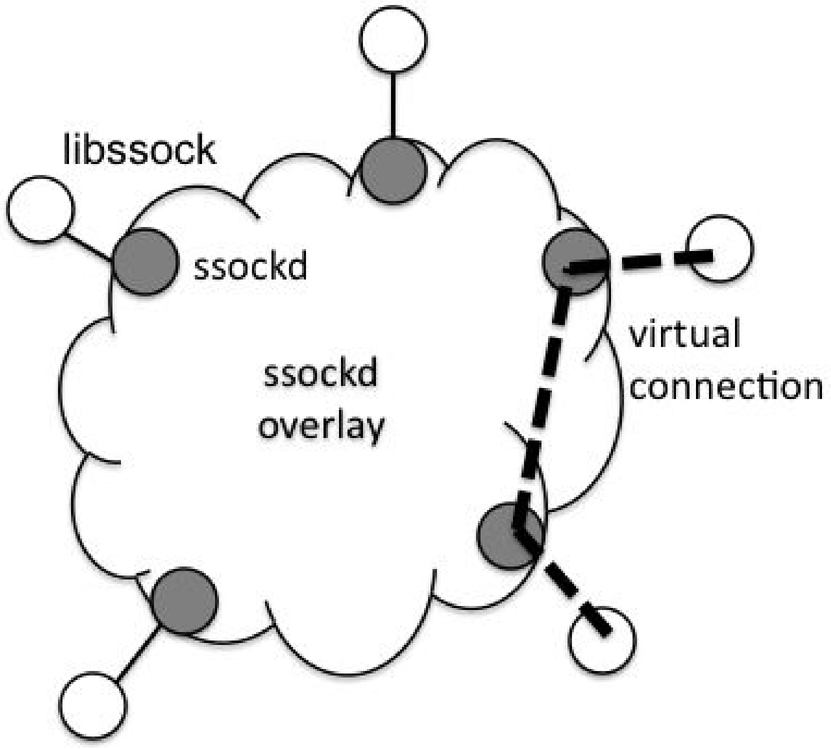


Figure 6.1: Overview of SSOCK

- Has comparable point-to-point performance as a regular Sockets library (which does not use a overlay network).
- Has better collective communication performance than a regular Sockets library.

Figure 6.1 illustrates the design of SSOCK. A forwarding daemon (ssockd) is brought up on each node, and the ssockds construct a locality-aware overlay network using the method described in Chapter 4.3. The application is linked to a communication library (libssock), which connects to the ssockd running on the same node. When the application calls `connect`, a virtual connection is established via the ssockd overlay, and when the application calls `send`, data is delivered via the virtual connection.

This design is similar to that of SmartSockets [41], but there are two important differences:

- Data is routed through the ssockd overlay even when direct communication is possible. This is crucial for achieving high scalability. Moreover, it improves collective communication performance, while only having a negligible impact on point-to-point communication performance.

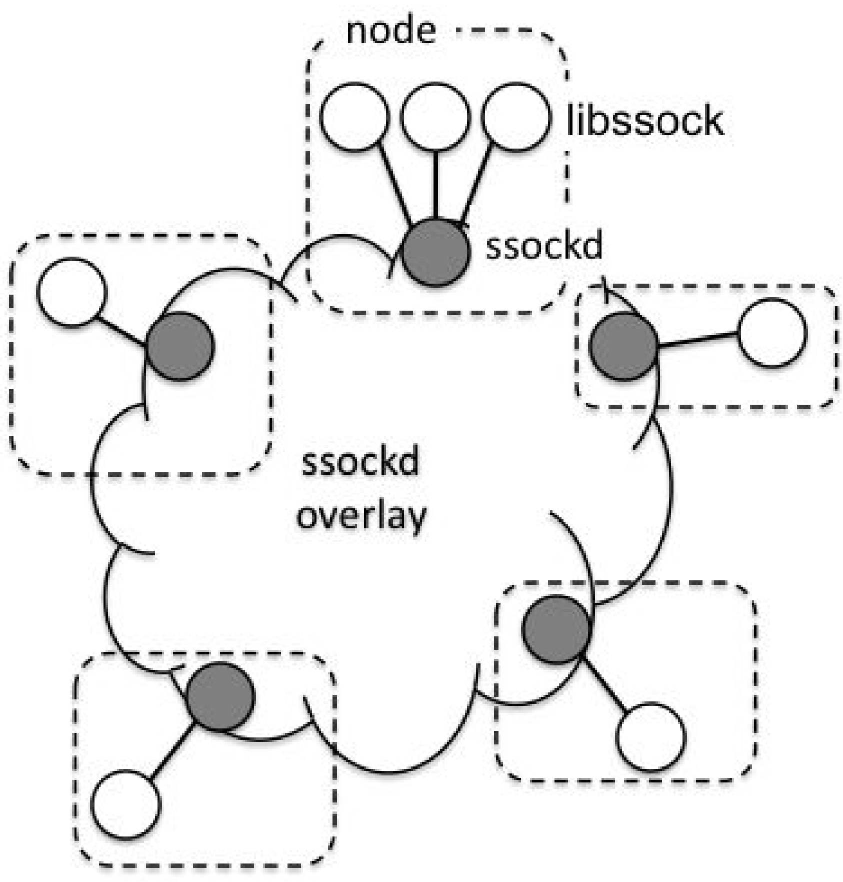


Figure 6.2: Libssocks and ssockds

- The number of processes that forward data to other clusters is not limited to one per cluster. This allows the inter-cluster bandwidth to be fully utilized, as was shown in the MC-MPI experiments in Section 5.6.

The rest of this section is organized as follows. In Subsection 6.1.2, I describe libssock, the library component of SSOCK. Then, in Subsection 6.1.3, I describe ssockd, the daemon component of SSOCK.

6.1.2 Libssock

Libssock is the library component of SSOCK, and is linked to applications that use SSOCK. It supports the primary parts of the Sockets API (i.e., `connect`, `bind`, `listen`, `accept`, `send` and `recv`). The operating system versions of these functions are overridden by preloading libssock. This is accomplished by using mechanisms such as `LD_PRELOAD` on Linux and

`DYLD_INSERT_LIBRARIES` on Mac OS X.

To the application, it appears as if endpoints are connected directly when `connect` is called and as if data is sent directly when `send` is called. Internally, however, endpoints are connected virtually via the overlay network constructed by ssockds, and data is sent using those virtual connections.

In order to access the ssockd overlay, upon the first invocation of a libssock function (usually `socket`), libssock connects to the ssockd running on the same node. When multiple libssocks are running on the same node (e.g., when multiple communicating processes are brought up on a multi-core node), all of the libssocks running on the same node connect to the same ssockd. SSOCK can easily be changed to allow a libssock to connect to an ssockd in a different node, but this option has not been explored yet. Figure 6.2 depicts the mapping between libssocks and ssockds.

6.1.3 Ssockd

Ssockd is the daemon component of SSOCK, and is responsible for transporting data among libssocks. The current implementation requires exactly one ssockd to be brought up on every node that will be used for computation, but this can be easily changed so that libssocks connect to ssockds running on different nodes as necessary.

The connection management scheme used by ssockd is basically that same as that used by MC-MPI. First, GXP [72] is used for endpoint exchange (Section 4.1), and latency between ssockds is measured using an estimation scheme (Subsection 4.2.1). Next, the latency information is used to construct a bounding graph consisting of temporary connections (Subsection 4.3.2), and a routing table is constructed using edges of the bounding graph (Subsection 4.3.3). Then, a spanning tree is created from the bounding graph, and all of the temporary connections except for those of a spanning tree are closed (Subsection 4.3.4). Finally, real connections are established lazily between neighboring processes of the bounding graph (Subsection 4.3.5).

6.2 Experimental results

6.2.1 Experimental setup

This section presents the results to the experiments that I performed to evaluate SSOCK. While SSOCK is still only a prototype implementation, these results show that SSOCK has many of

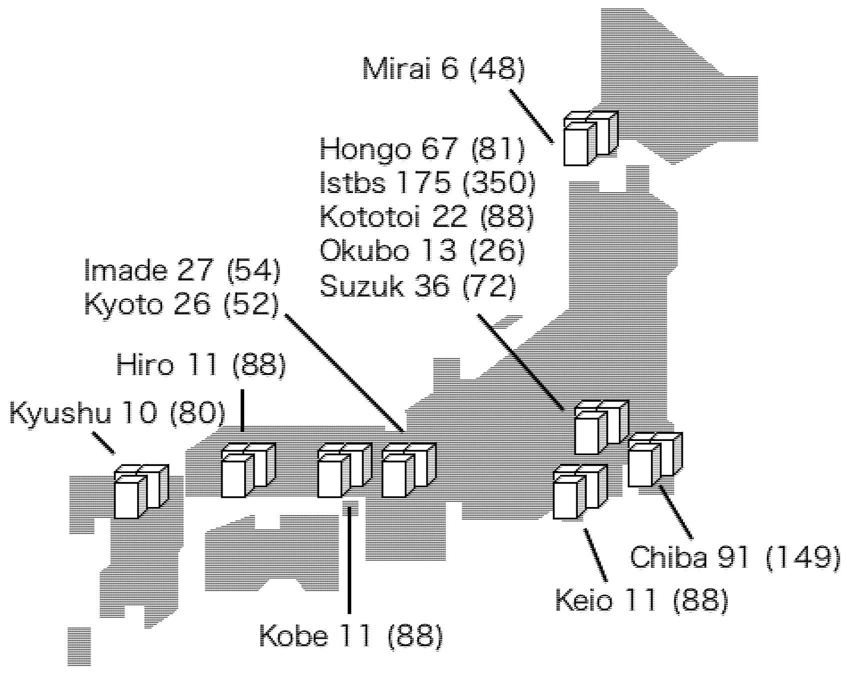


Figure 6.3: Experimental environment

the properties of MC-MPI as well as some new ones specific to Sockets libraries.

The experiments were performed using the environment shown in Figure 6.3. The environment consisted of 13 Linux clusters located in different parts of Japan and had a total of 506 nodes (1,264 cores). The network configuration of each cluster is shown in Table 6.1. Here, Firewall, Global and NAT denote the following:

Firewall: Nodes in this configuration had global IP addresses, but incoming connects from Istbs and Kototoi were filtered. All other traffic, including outgoing connects to Kototoi and Istbs, was not filtered.

Global: Nodes in this configuration had global IP addresses, and traffic was not filtered.

NAT: Nodes in this configuration only had private IP addresses. A multi-homed gateway performed address translation for these nodes, but the gateway itself was not used in my experiments.

Table 6.1: Network configuration of each cluster

Cluster	Network
Chiba	Global
Hiro	Global
Hongo	Global
Imade	NAT
Istbs	Global
Keio	Global
Kobe	Firewall
Kototoi	Global
Kyoto	NAT
Kyushu	Global
Mirai	Global
Okubo	Global
Suzuk	Global

6.2.2 Connectivity and scalability

In this subsection, I show that SSOCK handles connectivity and scalability issues better than the regular Sockets library as well as SmartSockets. A simple test was used for this purpose: bring up a process on each core and establish connections between all pairs of processes. In order to avoid SYN packets from being dropped, connection attempts were coordinated so that they only occurred one at a time (simultaneous connects are discussed in Subsection 6.2.3).

As expected, the regular Sockets library was not able to complete the test due to connectivity problems:

- Connections could not be established between the NAT clusters (Imade and Kyoto) regardless of the direction.
- Connections could not be established to the NAT clusters from the other 11 clusters.
- Connections could not be established to the Firewall cluster (Kobe) from Istbs and Kototoi.

Meanwhile, a SmartSockets-like implementation (SmartSockets itself was not publicly available) suffered from no connectivity problems:

- Connections within clusters and between Global clusters were established using the Direct technique. Connections from the NAT clusters and the Firewall cluster to the Global clusters were also established using the Direct technique.

- Connections between the NAT clusters were established using the Routed technique.
- Connections to the NAT clusters from the other 11 clusters were established using the Reverse technique. Connections from Istbs and Kototoi to the Firewall cluster were also established using the Reverse technique.

However, the SmartSockets-like implementation suffered from two scalability problems:

- The limit on the number of file descriptors that each process could use was reached. Each process had to connect to 1,263 other processes, but the operating system limit on the number of file descriptors was 1,024. I was able to overcome this limitation by increasing this limit on every node, but this may not always be possible because it requires administrative privileges.
- Even after increasing the limit on the number of file descriptors, another resource allocation problem was encountered: the limit on the number of connections that could be handled by a single NAT gateway. The Direct and Reverse techniques tried to establish a total of 57,780 connections through Imade's gateway, but the gateway's NAT table became full after 53,800 connections were established.

When SSOCK was used with the percentage of connections limited to 30%, connections were established between all 1,264 processes without any of the previously mentioned connectivity or scalability problems; connectivity issues were solved by the ssockd overlay, and scalability issues were solved by limiting the number of connections.

6.2.3 Simultaneous connects

In my next experiment, I compared the behaviors of SSOCK and SmartSockets-like when a large number of connection establishment attempts were made simultaneously. The same number of processes was brought up in each cluster, and every process tried to connect to every other process simultaneously using non-blocking connects.

Figure 6.4 shows the results for 13 to 338 processes (1 to 26 processes per cluster). With SmartSockets-like, it took 38 seconds to completely connect 52 processes, and the operation failed for 78 or more processes because some connection attempts did not complete within 189 seconds and timed out [1]. With SSOCK, it only took 1.8 seconds to completely connect 52

[1]In Linux, the `connect` system call times out after 189 seconds.

processes (10.2 seconds if the 8.4 seconds required to construct the ssockd overlay is included), and less than 10 seconds even for 338 processes.

SmartSockets-like performed so poorly, because many SYN packets were dropped by routers. Routers often prevent a large number of SYN packets from passing through them for security reasons. As a result, the number of simultaneous connection attempts needs to be controlled in order to avoid packet losses. This slows down the connection establishment phase and it also makes programming difficult.

Meanwhile, SSOCK did not suffer from packet losses, because the number of SYN packets that passes through routers was small. The establishment of some virtual connections caused the lazy establishment of real connections, but the number of real connections established was limited (to 30%), and most of them were between nearby ssockds. Therefore, unlike SmartSockets and other simplistic schemes that establish a large number of wide-area connections, SSOCK does not require connection establishment attempts to be "paced."

6.2.4 Point-to-point and collective communication performance

In this subsection, I show that the point-to-point communication performance of SSOCK is comparable to that of SmartSockets-like, and that the collective communication performance of SSOCK is better than that of SmartSockets-like. For all of the experiments in this subsection, a 240-node ssockd overlay was used. 240 was the number of nodes necessary for using 338 cores (26 cores in each of the 13 clusters).

First, the intra-cluster point-to-point communication performance was measured by performing the ping-pong test using two nodes from Kototoi. As shown in Figure 6.5, SSOCK and SmartSockets-like had similar performance. SSOCK performed as well as SmartSockets-like despite communicating via the ssockd overlay, because the overlay was constructed in a locality-aware manner. Messages traveled from one ping-pong process (libssock) to the ssockd running on the same node, then to the ssockd running on the other node, and finally to the other ping-pong process (libssock). Thus, compared to SmartSockets-like, SSOCK involved some extra communication, but this did not lower the throughput because the extra communication was contained within a single node.

Next, the inter-cluster point-to-point communication performance was measured by performing the ping-pong test using one node from Hongo and one node from Okubo. Once again, as shown in Figure 6.6, SSOCK and SmartSockets-like had similar performance. This time, when SSOCK was used, messages traveled through three ssockds instead of two (the third ssockd was in Hongo but a node different from the one where the ping-pong process was running). However, the negative impact on performance was negligible, because the extra hop inside Hongo had more bandwidth than the inter-cluster hop.

Finally, collective communication performance was measured by performing the all-to-all operation (the equivalent of `MPI_Alltoall` but programmed with `send` and `recv`). As shown in Figure 6.7, SSOCK consistently performed better than SmartSockets-like. The reason for this is that SSOCK limited the number of connections that each process established, which as in the case of Integer Sort (IS) in the NAS Parallel Benchmarks (NPB) [13, 75] and Distributed Verification Environment (DiVinE) [2, 28], helped reduced congestion (Section 5.6).

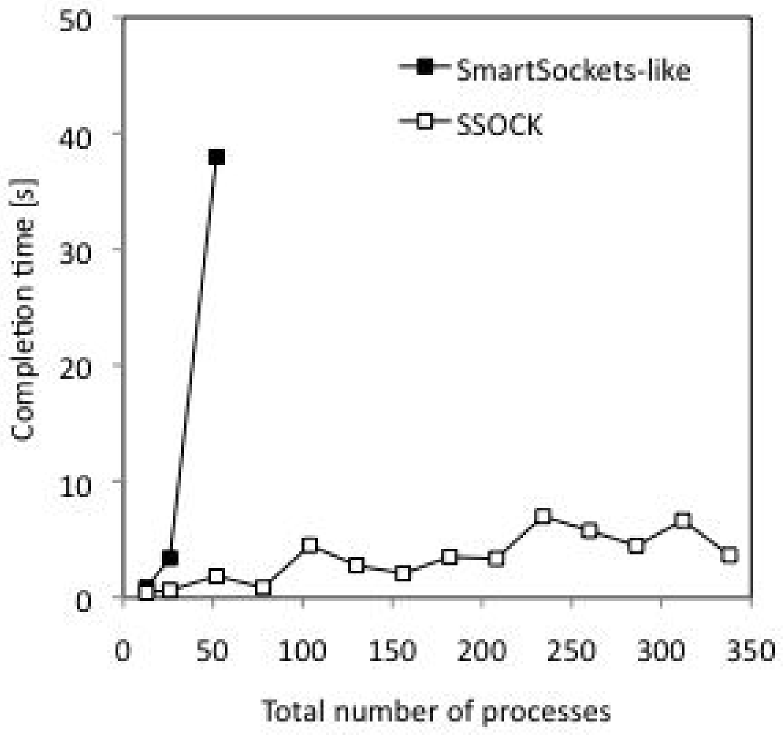


Figure 6.4: Completion time of all-to-all connect

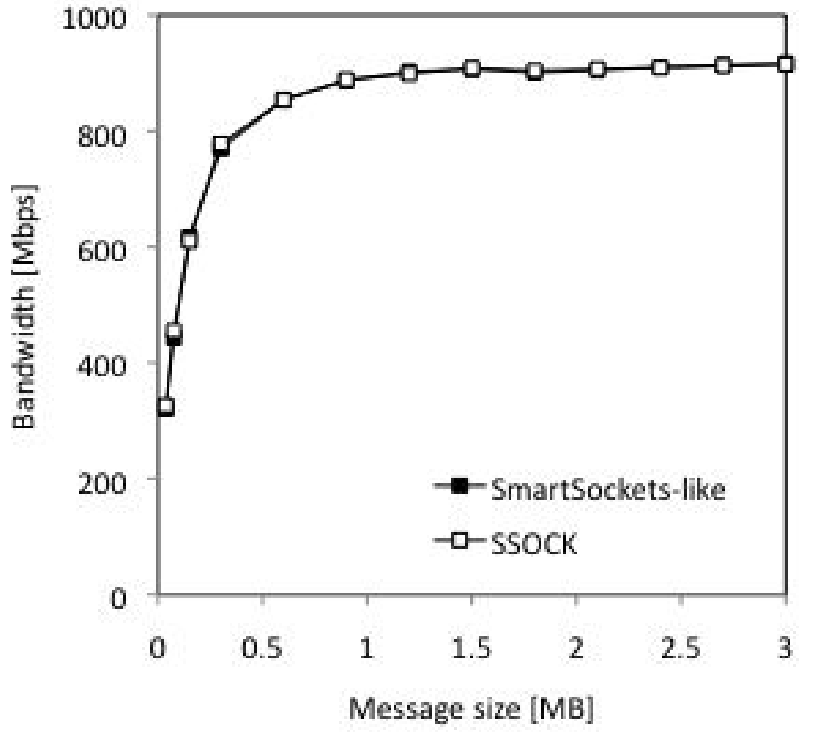


Figure 6.5: Intra-cluster ping-pong performance

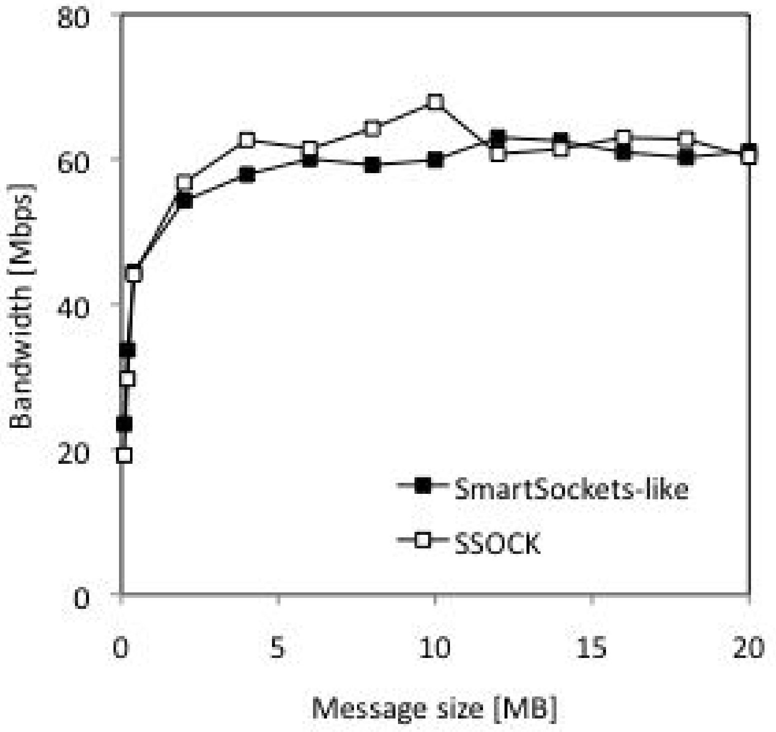


Figure 6.6: Inter-cluster ping-pong performance

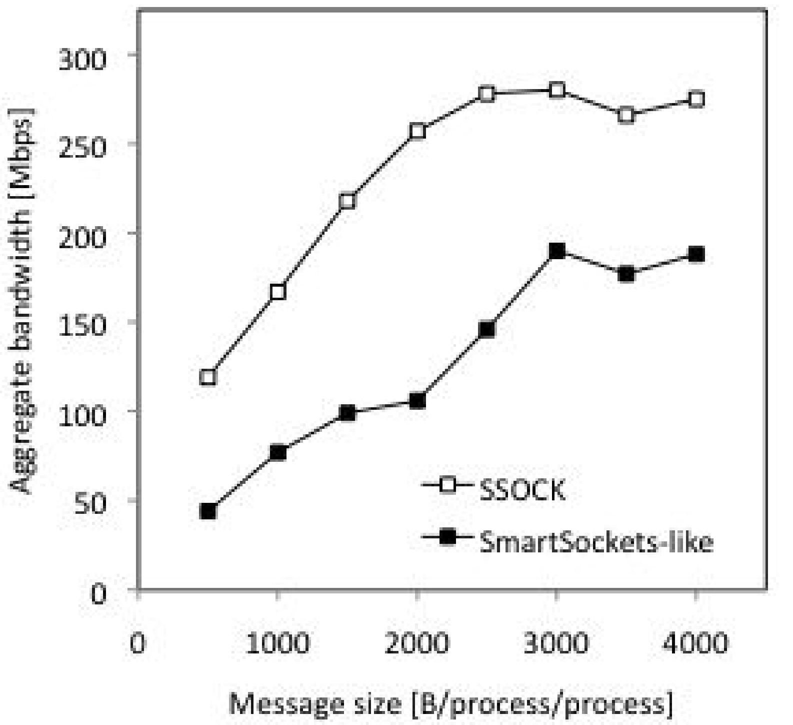


Figure 6.7: All-to-all performance

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation, I studied the design and implementation of scalable high-performance communication libraries for wide-area computing environments. I began by discussing the increase in bandwidth of WANs and the new possibilities that it brought to parallel computation using multi-cluster environments. Unfortunately, multi-cluster environments are significantly more complex than single cluster environments, and they introduce or magnify problems concerning connectivity, scalability and locality. After describing those problems, I discussed related work in a general context and in the context of message passing, and pointed out the shortcomings of existing solutions. In order to overcome those shortcomings, I made two proposals for wide-area communication libraries: a locality-aware connection management scheme and a locality-aware rank assignment scheme.

My connection management scheme overcomes firewalls and NAT by constructing an overlay network, and achieves scalability by limiting the number of connections that each process establishes to $O(\log n)$ when the total number of processes is n. In order to achieve high performance with a limited number of connections, the connections that are established are selected in a locality-aware manner, based on latency and traffic information obtained from a short profiling run.

Meanwhile, my rank assignment scheme finds a low-overhead mapping between ranks and processes by formulating the rank assignment problem as a QAP. It adapts to the environment and to the application by setting up the QAP using the latency and traffic information obtained from the profiling run. An approximate solution to the QAP is found using a library that uses the GRASP heuristic.

Using my proposed connection management and rank assignment schemes, I implemented a wide-area MPI library called MC-MPI, and evaluated its performance using 256 cores distributed equally across 4 clusters. Highlights of these experiments included the following:

- MC-MPI was able to limit the percentage of connections that each process established to as low as 10 percent, while performing at least as well as existing methods.
- For benchmarks in which many processes communicated simultaneously (IS and DiVinE), MC-MPI actually performed better than when many connections were used.
- MC-MPI was able to find rank assignments that performed up to 1.2 times better than commonly used assignments based on host names and up to 4.0 times better than locality-unaware assignments.

In order to increase the number of applications that are supported, I used my connection management scheme to implement a wide-area Sockets library called SSOCK. Highlights of the experiments performed to evaluate SSOCK included the following:

- In a 13-cluster environment with firewalls and NAT, SSOCK was able to connect 1,262 processes with each other without any of the connectivity issues and resource allocation problems that were encountered by existing methods.
- In another experiment in which many processes simultaneously tried to establish connections with each other, SSOCK was able to quickly establish connections between all pairs of processes, while an existing method (SmartSockets-like) suffered from a large number of packet losses and finally timed out.
- The point-to-point performance of SSOCK was comparable to that of SmartSockets-like, while the collective communication performance of SSOCK was better than that of SmartSockets-like.

7.2 Future work

7.2.1 Connection management

A shortcoming of my connection management scheme is that it does not guarantee that bounding graphs will be connected. While I have shown through simulations that the disconnect probability of bounding graphs is low enough for practical use, a formal analysis is necessary for a deeper understanding of the properties of bounding graphs. Moreover, it is important to note that simply having a “connected” bounding graph is not enough for achieving high communication performance. Thus, one topic of particular interest is communication performance when connectivity of the underlying network is severely limited and the number of connections established by my connection management scheme deviates greatly from $O(\log n)$ connections per process.

7.2.2 SSOCK

One important task concerning SSOCK is to make it support all or as much as possible of the Sockets API. By doing so, all parallel applications instead of just MPI applications can take advantage of my connection management scheme. An interesting experiment would be to compare the performance of MC-MPI to that of MPICH or MPICH-2 linked to SSOCK.

Another task concerning SSOCK is to allow certain libssock pairs to establish real connections with each other directly instead of establishing virtual connections with each other through the ssockd overlay network. Candidates for such behavior include libssocks that communicate through one ssockd (intra-node communication) and libssocks that communicate through two ssockds (communication is direct at the node level). This would lower forwarding overhead without sacrificing scalability.

Yet another task is to allow a libssock to connect to an ssockd running on a different node. This would allow new nodes to be added to a computation without reconstructing the ssockd overlay network.

7.2.3 Broader goal

A broader goal is to study connection management and rank assignment schemes that recognize that there is a large variance in the bandwidth of wide-area links. My current schemes correctly recognize that the wide-area links are the weakest links in a multi-cluster environment, but fail to recognize the difference between wide-area links that are relatively wide (e.g., 10Gbps) and those that are relatively narrow (e.g., 100Mbps). One reason why such links need to be differentiated is that the number of forwarding processes should be small for narrow links while they should be large for wide links. Another reason is that rank assignment schemes should assign more traffic to wide links than to narrow links. Recognizing the variance in the bandwidth of links is expected to become more and more important as newer technologies make the wider links wider and wider.

Bibliography

[1] O. Aumage and G. Mercier. MPICH/MADIII: A Cluster of Clusters Enabled MPI Implementation. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 26–33, 2003.

[2] J. Barnat, L. Brim, I. Cerna, P. Moravec, P. Rockai, and P. Simecek. DiVinE – A Tool for Distributed Verification. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*, pages 278–281, 2006.

[3] M. Bazarra and H. Sherali. On the Use of Exact and Heuristic Cutting Plane Methods for the Quadratic Assignment Problem. *Journal of the Operational Research Society (JORS)*, 33:991–1003, 1982.

[4] G. Bhanot, D. Chen, A. Gara, and P. Vranas. The Blue Gene/L Supercomputer. *Nuclear Physics B – Proceedings Supplements*, 119:114–121, 2003.

[5] G. Bhanot, A. Gara, P. Heidelberger, E. Lawless, J. Secton, and R. Walkup. Optimizing Task Layout on the Blue Gene/L Supercomputer. *IBM Journal of Research and Development*, 49(2/3):489–500, 2005.

[6] R. Blumofe and C. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[7] Brunet Software Library. http://boykin.acis.ufl.edu/wiki/index.php/Brunet.

[8] R. Burkard and T. Bonniger. A Heuristic for Quadratic Boolean Program with Applications to Quadratic Assignment Problems. *European Journal of Operational Research (EJOR)*, 13:374–386, 1983.

[9] R. Burkard, S. Karisch, and F. Rendl. QAPLIB - A Quadratic Assignment Problem Library. *Journal of Global Optimization (JGO)*, 10:391–403, 1997.

[10] R. Burkard and F. Rendl. A Thermodynamically Motivated Simulation Procedure for Combinatorial Optimization Problems. *European Journal of Operational Research (EJOR)*, 17:169–174, 1984.

[11] K. Chin, J. Judge, A. Williams, and R. Kermode. Implementation Experience with MANET Routing Protocols. *ACM SIGCOMM Computer Communications Review (CCR)*, 32(5):49–59, 2002.

[12] A. Denis, O. Aumage, R. Hofman, K. Verstoep, T. Kielmann, and H. Bal. Wide-area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems. In *Proceedings of the 13th International Symposium on High-Performance Distributed Computing (HPDC)*, pages 97–106, 2004.

[13] R. der Wijngaart. NAS Parallel Benchmarks Version 2.4. NAS Technical Report NAS-02-007, NASA Ames Research Center, 2002.

[14] E. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] R. Dube, C. Rais, K. Wang, and S. Tripathi. Signal Stability-Based Adaptive Routing (SSA) for Ad Hoc Mobile Networks. *IEEE Personal Communications (PCM)*, 4(1):36–45, 1997.

[16] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed Computing in a Heterogeneous Computing Environment. In *Proceedings of the 5th European PVM/MPI User's Group Meeting*, pages 180–187, 1998.

[17] A. Ganguly, A. Agrawal, P. Boykin, and R. Figueiredo. IP over P2P: Enabling Self-configuring Virtual IP Networks for Grid Computing. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[18] A. Ganguly, A. Agrawal, P. Boykin, and R. Figueiredo. WOW: Self-Organizing Wide Area Overlay Networks of Virtual Workstations. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 30–41, 2006.

[19] T. Goff, N. Abu-Ghazaleh, D. Phatak, and R. Kahvecioglu. Preemptive Routing in Ad Hoc Networks. In *Proceedings of the Seventh ACM SIGMOBILE Annual International Conference on Mobile Computing and Networking (MobiComm)*, pages 43–52, 2001.

[20] S. Gorlatch. Send-Receive Considered Harmful: Myths and Realities of Message-Passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, 2004.

[21] Grid'5000. http://www.grid5000.fr/.

[22] GridMPI. http://www.gridmpi.org/.

[23] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[24] S. Guha and P. Francis. Characterization and Measurement of TCP Traversal Through NATs and Firewalls. In *Proceedings of the 2005 Internet Measurement Conference (IMC)*, pages 199–211, 2005.

[25] S. Guha, Y. Takeda, and P. Francis. Nutss: A SIP-based Approach to UDP and TCP Network Connectivity. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, pages 43–48, 2004.

[26] T. Hatazaki. Rank Reordering Strategy for MPI Topology Creation Functions. In *Proceedings of the 5th European PVM/MPI User's Group Meeting*, pages 188–195, 1998.

[27] http://anna.fi.muni.cz/models/. BEEM: BEnchmarks for Explicit Model checkers.

[28] http://divine.fi.muni.cz/. DiVinE Homepage.

[29] http://www.softether.com/. SoftEther Corporation (in Japanese).

[30] C. Huang, O. Lawlor, and L. Kale. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 306–322, 2003.

[31] C. Huang, G. Zheng, S. Kumar, and L. Kale. Performance Evaluation of Adaptive MPI. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 12–21, 2006.

[32] InTrigger. http://www.intrigger.jp/.

[33] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 377–384, 2000.

[34] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, 2003.

[35] G. Karypis and V. Kumar. Multilevel Graph Partitioning Schemes. In *Proceedings of the 24th International Conference on Parallel Processing (ICPP)*, pages 113–122, 1995.

[36] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[37] T. Kielmann, R. Hofman, H. Bal, A. Plaat, and R. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 131–140, 1999.

[38] T. Koopmans and M. Beckman. Assignment Problems and the Location of Economic Activities. *Econometrica*, 25:53–76, 1957.

[39] LAN/MAN Standards Committee, IEEE Computer Society. IEEE Std. 802.3-2005 Standard for Information Technology – Telecommunications and Information Exchange Between Systems – Local and Metropolitan Area Networks – Specific Requirements, 2005.

[40] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. IETF RFC 1928, 1996.

[41] J. Maassen and H. Bal. SmartSockets: Solving the Connectivity Problems in Grid Computing. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 1–10, 2007.

[42] Message Passing Interface Forum. http://www.mpi-forum.org/.

[43] A. Misevicius. A Modified Simulated Annealing Algorithm for the Quadratic Assignment Problem. *Informatica*, 14(4):497–514, 2003.

[44] A. Misevicius. A Fast Hybrid Genetic Algorithm for the Quadratic Assignment Problem. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1257–1264, 2006.

[45] MPICH – A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich1/.

[46] MPICH-G2. http://www3.niu.edu/mpi/.

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

[48] Next-generation Science Information Network SINET3. http://www.sinet.ad.jp/.

[49] METIS Family of Multilevel Partitioning Algorithms. http://www.cs.umn.edu/˜metis/.

[50] OpenVPN. http://openvpn.net/.

[51] R. Pelanek. Web Portal for Benchmarking Explicit Model Checkers. Tech. Report FIMU-RS-2006-03, Masaryk University Brno, 2006.

[52] C. Plaxton, R. Rajaraman, and A. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 311–320, 1997.

[53] Portable Applications Standards Committee, IEEE Computer Society. IEEE Std. 1003.1-2004 Standard for Information Technology – Portable Operating System Interface (POSIX), 2004.

[54] J. Postel. User Datagram Protocol. IETF RFC 768, 1980.

[55] J. Postel. Transmission Control Protocol. IETF RFC 793, 1981.

[56] QAPLIB – A Quadratic Assignment Problem Library. http://www.opt.math.tu-graz.ac.at/qaplib/.

[57] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM)*, pages 161–172, 2001.

[58] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). IETF RFC 4271, 2006.

[59] M. Resende and P. Pardalos. Algorithm 754: Fortran Subroutines for Approximate Solution of Dense Quadratic Assignment Problems Using GRASP. *ACM Transactions on Mathematical Software (TOMS)*, 22(1):104–118, 1996.

[60] E. Rosen and Y. Rekhter. BGP/MPLS VPN. IETF RFC 2547, 1999.

[61] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. IETF RFC 3031, 2001.

[62] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). IETF RFC 3489, 2003.

[63] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[64] TOP500 Supercomputing Sites. http://www.top500.org/.

[65] J. Snader. *Effective TCP/IP Programming: 44 Tips to Improve Your Network Programs*. Addison Wesley, 2000.

[66] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). IETF RFC 3022, 2001.

[67] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. IETF RFC 2663, 1999.

[68] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (SIGCOMM)*, pages 149–160, 2001.

[69] Surfnet. http://www.surfnet.nl/.

[70] R. Takano, M. Matsuda, T. Kudoh, Y. Kodama, F. Okazaki, and Y. Ishikawa. Effects of Packet Pacing for MPI Programs in a Grid Environment. In *Proceedings of the 9th IEEE International Conference on Cluster Computing (Cluster)*, pages 382–391, 2007.

[71] R. Takano, M. Matsuda, T. Kudoh, Y. Kodama, F. Okazaki, Y. Ishikawa, and Y. Yoshizawa. IMPI Relay Trunking for Improving the Communication Performance on Private IP Clusters. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 401–408, 2008.

[72] K. Taura. GXP: An Interactive Shell for the Grid Environment. In *Proceedings of the 8th International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA)*, pages 59–67, 2005.

[73] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH. In *Proceedings of the 10th European PVM/MPI User's Group Meeting*, pages 257–267, 2003.

[74] The Distributed ASCI Supercomputer 3. http://www.cs.vu.nl/das3/.

[75] The NAS Parallel Benchmarks. http://www.nas.gov/Software/NPB/.

[76] J. Träff. Implementing the mpi process topology mechanism. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC)*, 2002.

[77] Virtual Bridged Local Area Networks. IEEE Std. 802.1Q-2005 Standard for Local and Metropolitan Area Networks, 2005.

[78] A. Woo, T. Tong, and D. Culler. Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 14–27, 2003.

[79] Y. Xu, M. Lim, Y. Ong, and J. Tang. A GA-ACO-Local Search Hybrid Algorithm for Solving Quadratic Assignment Problem. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 599–605, 2006.

[80] T. Ylonen. The Secure Shell (SSH) Authentication Protocol. IETF RFC 4252, 2006.

[81] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications (J-SAC)*, 22(1):41–53, 2004.

Publications

Journal and Transaction Papers

[1] Hideo Saito, Kenjiro Taura and Takashi Chikayama. Collective Operations for Wide-area Message Passing Systems Using Adaptive Spanning Trees. *International Journal of High Performance Computing and Networking (IJHPCN).* Vol.5, No.3, 2008, pp.179–188.

[2] Hideo Saito and Kenjiro Taura. Locality-aware Connection Management and Rank Assignment for Wide-area MPI. *IPSJ Transactions on Advanced Computing Systems.* Vol.48 No.SIG 18 (ACS 20), pp.44–55, December 2007 (in Japanese).

[3] Hideo Saito, Kenjiro Taura and Takashi Chikayama. Collective Operations for Wide-area Message Passing Systems Using Adaptive Spanning Trees. *IPSJ Transactions on Advanced Computing Systems.* Vol.46 No.SIG 12 (ACS 11), pp.373–383, August 2005 (in Japanese).

[4] Hideo Saito, Kenjiro Taura and Takashi Chikayama. Expedite: An Operating System Extension to Support Low-latency Communication in Non-dedicated Clusters. *IPSJ Transactions on Advanced Computing Systems.* Vol.45 No.SIG 12 (ACS 7), pp.229–237, October 2004.

International Symposiums and Workshops

[1] Hideo Saito and Kenjiro Taura. Locality-aware Connection Management and Rank Assignment for Wide-area MPI. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007),* pp.249–256, Rio de Janeiro, Brazil, May 2007.

[2] Hideo Saito, Kenjiro Taura and Takashi Chikayama. Collective Operations for Wide-area Message Passing Systems Using Adaptive Spanning Trees. In *Proceedings of the 6th*

IEEE/ACM International Workshop on Grid Computing (Grid 2005), pp.40–48, Seattle, USA, November 2005.

Domestic Refereed Papers

[1] Hideo Saito and Kenjiro Taura. Locality-aware Connection Management and Rank Assignment for Wide-area MPI. In *Proceedings of the Symposium on Advanced Computing Systems and Infrastructures (SACSIS 2007),* pp.339–348, Tokyo, Japan, May 2007 (in Japanese).

[2] Hideo Saito, Kenjiro Taura and Takashi Chikayama. Expedite: An Operating System Extension to Support Low-latency Communication in Non-dedicated Clusters. In *Proceedings of the Symposium on Advanced Computing Systems and Infrastructures (SACSIS 2004),* pp.443–450, Sapporo, Japan, May 2004 (in Japanese).

Unrefereed Papers

[1] Hideo Saito, Kenjiro Taura. A Scalable High-performance Communication Library for Wide-area Environments. *IPSJ SIG Technical Report HPC-116 (SWoPP 2008),* pp.181–186, Saga, Japan, August 2008 (in Japanese).

[2] Hideo Saito, Yoshikazu Kamoshida, Shogo Sawai, Ken Hironaka, Kei Takahashi, Takeshi Sekiya, Nan Dun, Takeshi Shibata, Daisaku Yokoyama and Kenjiro Taura. InTrigger: A Multi-site Distributed Computing Environment Supporting Flexible Configuration Changes. *IPSJ SIG Technical Report HPC-111 (SWoPP 2007),* pp.237–242, Asahikawa, Japan, August 2007 (in Japanese).

[3] Hideo Saito, Kenjiro Taura and Takashi Chikayama. MPI/GXP: An Adaptive Message Passing System for Wide-area Environments. *IPSJ SIG Technical Report HPC-107 (SWoPP 2006),* pp.25–30, Kochi, Japan, July 2006 (in Japanese).

[4] Hideo Saito, Kenjiro Taura and Takashi Chikayama. Latency-aware Connection Management for Wide-area Message Passing Systems. *IPSJ SIG Technical Report HPC-103 (SWoPP 2005),* pp.181–185, Takeo, Japan, August 2005 (in Japanese).

[5] Hideo Saito, Kenjiro Taura and Takashi Chikayama. Collective Operations for the Phoenix Programming Model. *IPSJ SIG Technical Report HPC-99 (SWoPP 2004),* pp.223-228, Aomori, Japan, July 2004 (in Japanese).

Poster Papers

[1] Hideo Saito, Ken Hironaka and Kenjiro Taura. A Scalable High-performance Communication Library for Wide-area Environments. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid 2008),* pp.310–315, Tsukuba, Japan, September 2008.

[2] Hideo Saito and Kenjiro Taura. Locality-aware Connection Management and Rank Assignment for Wide-area MPI. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2007)*, pp.150–151, San Jose, USA, March 2007.

[3] Hideo Saito, Kenjiro Taura and Takashi Chikayama. MPI/GXP: An Adaptive Message Passing System for Wide-area Environments. At *Symposium on Advanced Computing Systems and Infrastructures (SACSIS 2006),* Osaka, Japan, June 2005 (in Japanese).

[4] Hideo Saito, Kenjiro Taura and Takashi Chikayama. Collective Operations for Wide-area Message Passing Systems Using Dynamic Spanning Trees. At *Symposium on Advanced Computing Systems and Infrastructures (SACSIS 2005),* Tsukuba, Japan, May 2005 (in Japanese).

Co-authored Papers

[1] Ken Hironaka, Hideo Saito, Kei Takahashi and Kenjiro Taura. A Framework for Flexible Programming in Complex Grid Environments. *IPSJ Transactions on Advanced Computing Systems (ACS).* Vol.1 No.2, pp.157–168, August 2008 (in Japanese).

[2] Tatsuya Shirai, Hideo Saito and Kenjiro Taura. A Fast Topology Inference—A Building Block for Network-aware Parallel Processing. *IPSJ Transactions on Advanced Computing Systems.* Vol.48 No.SIG 13 (ACS 19), pp.156–165, August 2007 (in Japanese).

[3] Ken Hironaka, Hideo Saito, Kei Takahashi and Kenjiro Taura. gluepy: A Simple Distributed Python Framework for Complex Grid Environments. At *the 21st Annual International Workshop on Languages and Compilers for Parallel Computing (LCPC 2008),* Edmonton, Canada, August 2008.

[4] Kei Takahashi, Hideo Saito and Kenjiro Taura. A Stable Broadcast Algorithm. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008)*, pp.392–400, Lyon, France, May 2008.

[5] Tatsuya Shirai, Hideo Saito and Kenjiro Taura. A Fast Topology Inference—A Building Block for Network-aware Parallel Processing. In *Proceedings of the 16th IEEE International Symposium on High Performance Distributed Computing (HPDC 2007)*, pp.11–21, Monterey, USA, June 2007.

[6] Takayoshi Shiraki, Hideo Saito, Yoshikazu Kamoshida, Katsuhiko Ishiguro, Ryo Fukano, Tatsuya Shirai, Kenjiro Taura, Mihoko Otake, Tomomasa Sato and Nobuyuki Otsu. Realtime Motion Recognition Using CHLAC Features and Cluster Computing. In *Proceedings of the 3rd IFIP International Conference on Network and Parallel Computing (NPC 2006)*, pp.50–56, Tokyo, Japan, October 2006.

Appendix

A.1 Quality of QAP solutions

Table A.1 lists the approximate solutions obtained by the library by Resende and Paradalos [59] for the 11 largest instances of the QAPLIB [9, 56]. n cores were used for problem size n (Table 5.1 shows the specifications of the compute nodes used). If the exact solution could not be found after 5 seconds, the search was stopped.

Table A.1: Solutions obtained by the QAP solver

Instance	n	Best known solution	Obtained solution	Gap
esc128	128	64	64	0%
sko100a	100	152002	152944	0.62%
sko100b	100	153890	154828	0.61%
sko100c	100	147862	149024	0.79%
sko100d	100	149576	150694	0.75%
sko100e	100	149150	150174	0.69%
sko100f	100	149036	150180	0.77%
tai100a	100	21052466	21563582	2.4%
tai256c	256	44759294	44879278	0.27%
tho150	150	8133398	8216878	1.0%
wil100	100	273038	273878	0.31%

A.2 Traffic matrices of benchmarks

Figure A.1 to A.7 give graphical representations of the traffic matrices of the seven benchmarks used in this dissertation. The gray areas represent rank pairs that perform a significant amount of communication, and the white areas represent rank pairs that perform little or no communication. Rank pairs that performed less than 1KB of communication during the profiling run are left white, as are rank pairs that performed less than 0.01% of the total communication.

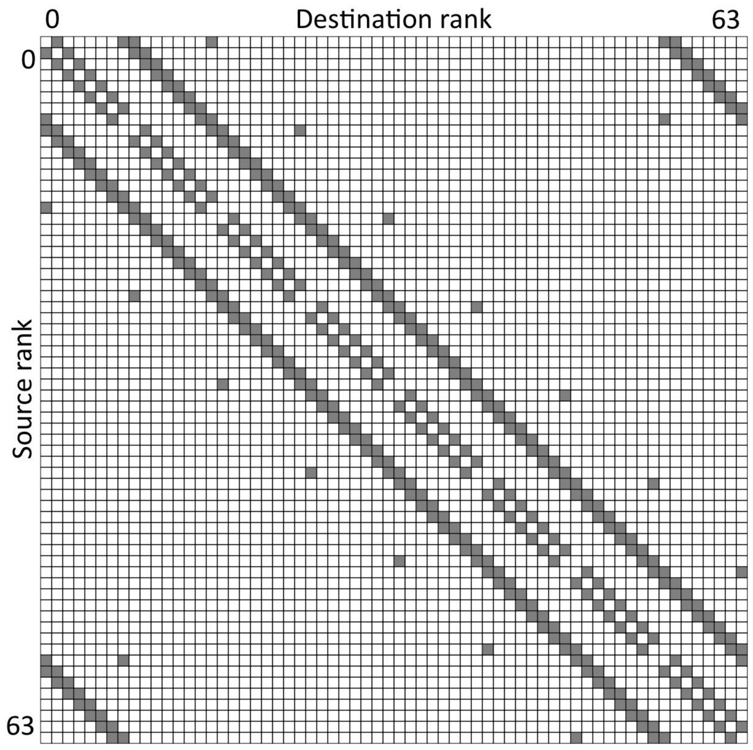


Figure A.1: Traffic matrix of BT (CLASS=D, NPROCS=64)

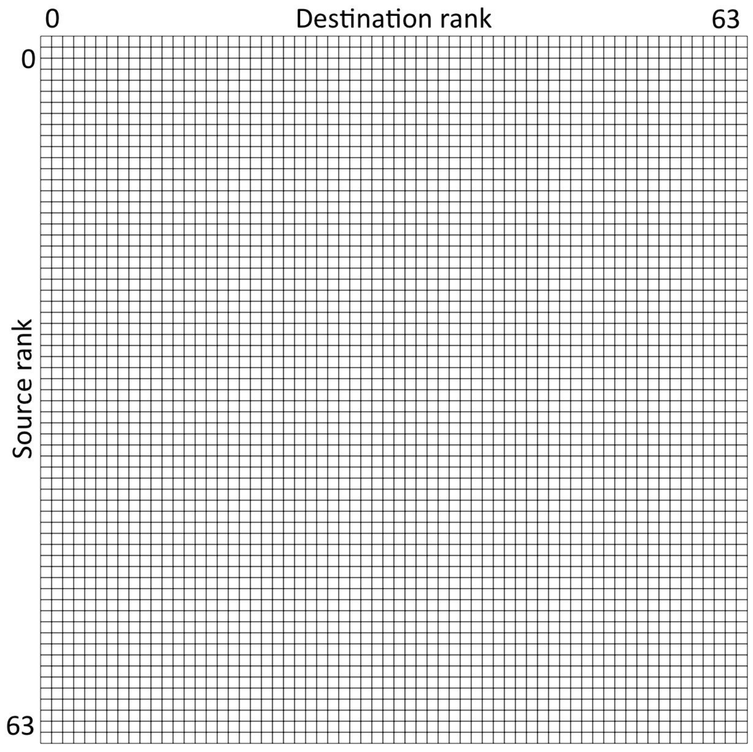


Figure A.2: Traffic matrix of EP (CLASS=D, NPROCS=64)

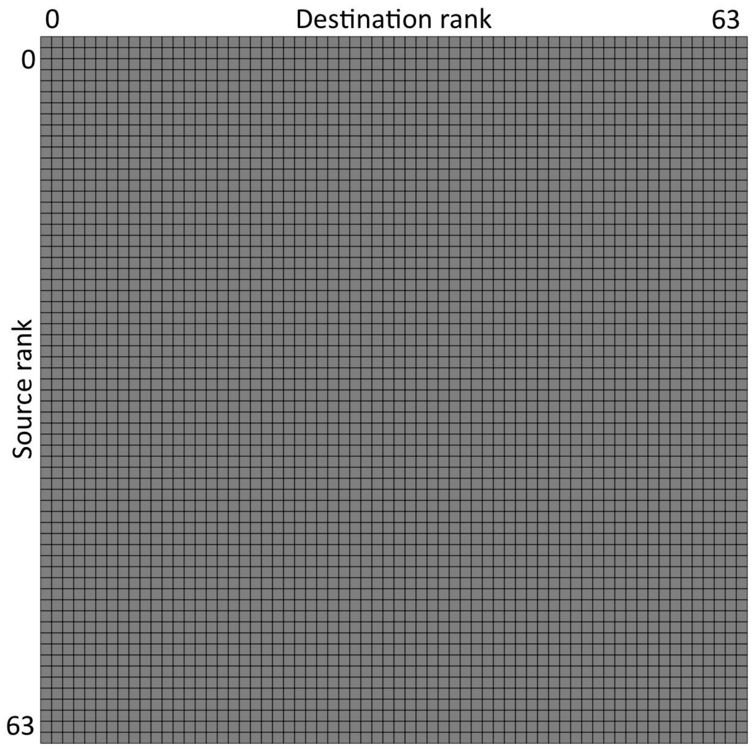


Figure A.3: Traffic matrix of IS (CLASS=C, NPROCS=64)

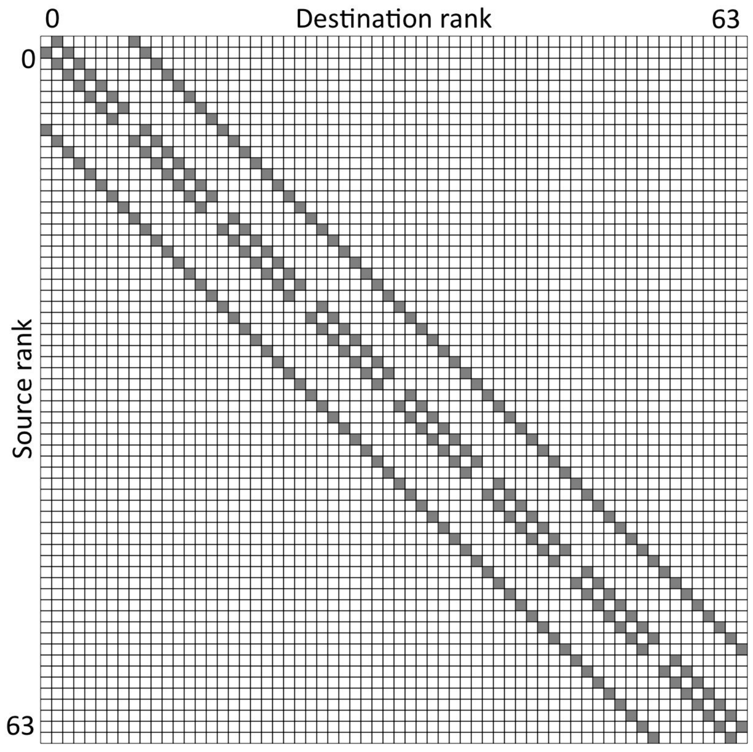


Figure A.4: Traffic matrix of LU (CLASS=D, NPROCS=64)

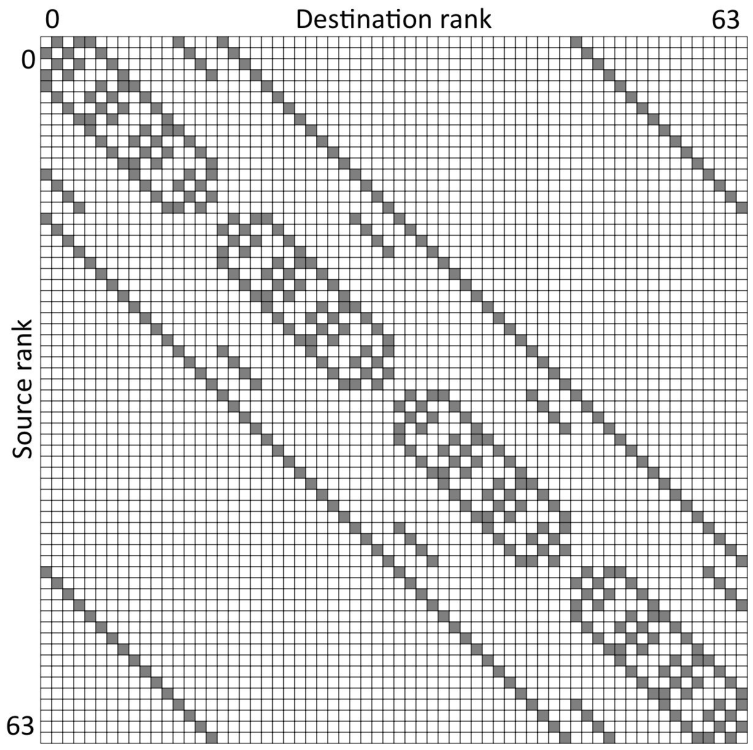


Figure A.5: Traffic matrix of MG (CLASS=D, NPROCS=64)

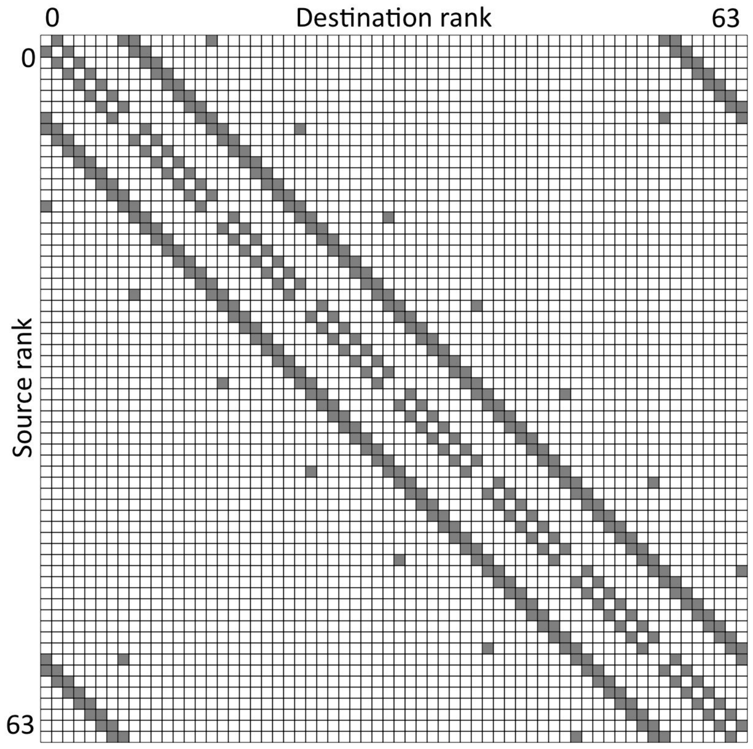


Figure A.6: Traffic matrix of SP (CLASS=D, NPROCS=64)

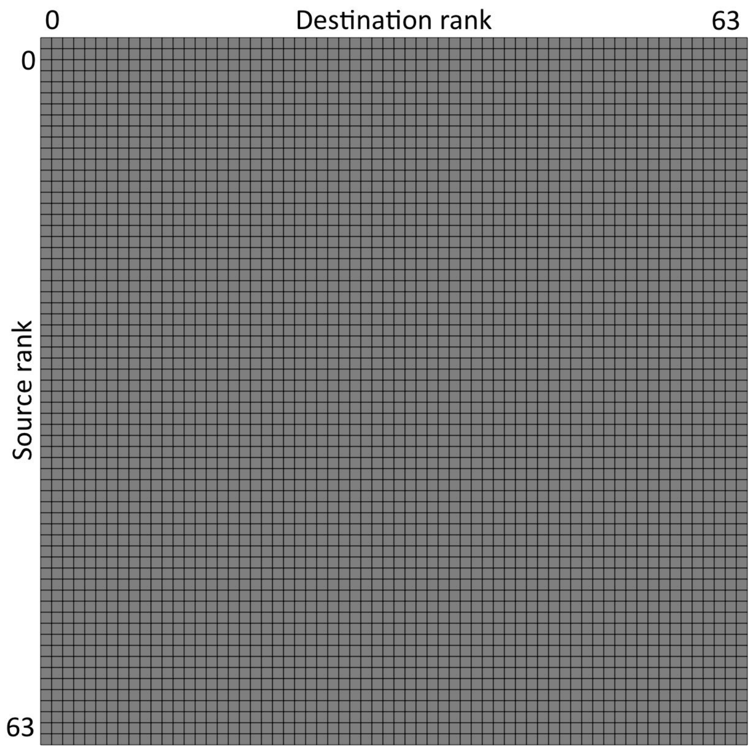


Figure A.7: Traffic matrix of DiVinE