Challenges of MPI over IPv6

Adrian Knoth, Christian Kauhaus, Dietmar Fey
Institute of Computer Science
Friedrich-Schiller-University Jena, Germany
{adi, kauhaus, fey}@inf.uni-jena.de

Lars Schneidenbach, Bettina Schnor
Institute of Computer Science
University Potsdam, Germany
{lschneid, schnor}@cs.uni-potsdam.de

Abstract

The Message Passing Interface (MPI) [17] is the most widely used message passing library for parallel applications on compute clusters. Here, we present our experiences in developing an IPv6 enabled MPI version for both most popular implementations, the MPICH2 and the Open MPI implementations. Further, we discuss how these IPv6 enabled MPI implementations can be used within multi-cluster and Grid topologies.

1. Introduction

The Message Passing Interface (MPI) is a specification for message-passing libraries [17]. It has become the de facto standard for high-performance parallel applications and is supported on a wide range of architectures, starting from clusters of PCs up to shared memory and vector machines. Various groups from industry and academia are working on MPI implementations. Several freely available implementations exist and, further, so called vendor MPI implementations exist, which are tuned for special hardware (see for example [9, 16, 5, 20, 3, 4]).

MPI specifies primitives for sending and receiving of messages. One of the reasons why it is so attractive compared to native socket programming is its simple addressing style. MPI applications address their communication partners with the pair (communicator, rank). All processes of an MPI application are in the same communicator COMM_WORLD. So, in spite of an IP address, a process only needs to know the rank of its communication partner. This makes writing of parallel applications much easier. More complex applications may use several communicators to divide the application processes into subgroups. Hence, an MPI application is not aware of IP addresses. This is hidden in the MPI libraries.

Previous work of two of the authors has confirmed that a good abstraction layer makes it very easy to adapt applications to network changes [21]. In that case study, they analysed the MPI-1 implementation MPICH from the Argonne National Lab. Due to the good design of MPICH where “[…] the ‘CH’ in MPICH stands for ‘Chameleon’, a symbol of adaptability to one’s environment and thus of portability” [9], this task was manageable with less effort.

But since then two major changes have affected the MPI implementations. First, the MPI implementers needed to implement also the new features from the MPI-2 standard [18], which has been postponed for a long time. Further, due to the need of petaflop computing, bigger and bigger systems are build. But running huge MPI applications on systems with ten thousands of processors has shown to be difficult. One main problem was the intolerable slow start-up time for process creation. Therefore, new MPI-2 implementations were developed with new process creation mechanisms. The most prominent open source implementations are MPICH2 from the Argonne National Lab and Open MPI. Open MPI started in 2004 and is maintained by a consortium of academic, research, and industry partners [8].

But why is there a need to enable these new MPI-2 implementations to support IPv6? The motivation is given by the Grid computing trend where several different compute sites are used to run parallel applications. For example, a user may want to run her applications distributed over different medium sized compute clusters within a university campus. The typical situation then is that the nodes within a cluster have private IP addresses, which makes inter-cluster communication impossible.

Up to now, two possible solutions have been proposed. Some systems realise inter-cluster communication with the help of user space daemons like for example PACX, Stampi, or MPICH/Madeleine [6, 10, 1]. But the major drawback of this approach is the performance loss for the communication [19, 14]. Another solution is to build a Virtual Private Network (VPN). But this means additional work for the system administrators, which is not very appreciated. The use of VPNs also has shown performance drawbacks [12, 14].

Another solution based on IPv6 addressing, announced already in the previous work [21], is presented here:
to enable IPv4 and IPv6 at the same time making MPICH support distributed computing across multiple clusters, where local communication is IPv4 and inter-cluster communication via the Internet uses IPv6. [21]

The aim is to support inter-cluster communication out-of-the-box without additional system administrator efforts and without performance loss. That this approach may be promising has shown related work where PVM-3.4.4, the predecessor of MPI, has been ported to IPv6 [15].

This paper investigates how two of the most popular MPI-2 implementations, MPICH2 and Open MPI, can be ported to work on top of an IPv6 network.

The remainder of the paper is structured as follows. The next section explains relevant details of MPICH2 and Open MPI. Section 3 presents the modifications applied to the implementations, and Section 4 shows measurement results of micro-benchmarks. The paper concludes in Section 5.

2. The design of MPI Implementations

To support different network protocols and hardware, both implementations use some kind of network abstraction layer. MPICH2 uses the Abstract Device Interface (ADI), while Open MPI network modules have to implement the Byte Transfer Layer (BTL). In this way the implementations are similar, but while the ADI is the central layer of MPICH2, the BTL is a so called component of the PML-framework. The BTL contains only low-level networking code and is controlled by the upper layer components of Open MPI.

Each MPI-implementation has to fulfil the task to map the abstract application level addresses (rank inside an MPI communicator) to a particular address that can be used by the network protocol. Usually, this is done during the initialisation of the runtime environment.

The initialisation of the runtime environment is the phase to obtain network addresses from the operating system to allow communication between the MPI processes. The kind of network and protocol has to be determined to choose the correct address type. Multiple communication paths can exist due to multiple network interfaces or multiple addresses per interface. Particular communication paths may not exist between some nodes, especially in case of multi-cluster environments. All these issues have to be solved before any mapping of ranks to addresses are possible. This makes the runtime environment a rather complex part of the implementations.

Since the network address handling is the concern of network specific code parts, the design and the implementation of the upper layers of both implementations are not important for the use of IPv6.

2.1. MPICH2

The central design component of MPICH2 is the Abstract Device Interface (ADI) to unify the network access by using a special layer to abstract the network. This interface is implemented by several so called channel devices and network modules to work on top of various network protocols (see Figure 1).

The most commonly implemented channel is the ch3-interface. Also a ch3-device called nemesis [2] is an implementation of this device. Nemesis is a recent development with the goal to achieve the best performance for shared memory based communication and improve the performance of communication via network.

Nemesis is designed in a modular way, so that different network protocols or hardware can be integrated by including network modules. The current adapter of the socket interface can handle IPv4 addresses and traffic only.

To shorten the time-consuming initialisation phase, MPICH2 uses a special runtime environment (see Figure 2). This environment basically consists of a daemon process called mpd (Multi Purpose Daemon). An mpd must run on each node and all daemons establish a connected logical ring to exchange address, state, and process information.

During the start-up of MPI applications, the mpd is responsible for creating a Process Manager (PM) daemon for each member of an MPI process group. The PM will hold host-port key pairs to provide the required information to establish connections between MPI processes. The Process Management Interface (PMI) is accessed by the local MPI process to retrieve network address information for communication purposes and to map a rank to network addresses.
3. The Architecture of MPI/IPv6

Both implementations have to make changes at different parts of the architecture to provide IPv6 capabilities: The network module inside the MPI-stack and the runtime environment. In the following, this is discussed separately for both implementations.

3.1. MPICH2

Moving MPICH2-nemesis towards IPv6, we took the first step by adding a new tcp6 module derived from the already existing tcp implementation. The porting of the module was straightforward, since only the creation of sockets and the address handling had to be adapted to IPv6. With a transparent usage of IPv4 and IPv6 in mind, the changes to the module include more generic socket-API calls that are more independent of a specific address family.

The second (and more complex) change affects the runtime environment, namely the mpd and PMI. The mpd code is modified in much the same way as the network module. The only exception is the usage of an unspecified address family that is not straightforward in the mpd-Python-code. Our intermediate solution opens listening sockets with IPv6 only. Therefore, transparent communication via IPv4 and IPv6 is currently not possible.

The interface between runtime environment and MPI-processes (PMI) had to be modified, too. The mpd is able to work with so called interface aliases that represent special interfaces and their address. In the original implementation, the alias was used to retrieve an IPv4 address. This is inadequate for IPv6. Addresses and ports are stored in the so called key value space (KVS in Figure 2) to provide contact information of other processes. In the KVS, the separator between address and port was a colon which is ambiguous with IPv6. To achieve a transparent behaviour, we use the fully qualified domain name instead of the address. Using the fully qualified domain names has a drawback: It depends on the configuration of the host names to work correctly or not. The resolver has to be configured in such a way that the best suited host address is returned first. Therefore, further efforts will be made to this implementation detail, until full transparency is achieved.

3.2. Open MPI

Adding IPv6 support to Open MPI affected all three abstraction layers: First, OPAL’s helper functions for socket-based connections were extended to deal with both address families, IPv4 and IPv6. Open MPI retrieves all local addresses by querying each kernel interface at runtime. Therefore, the new OPAL interface discovery also searches for IPv6 addresses and propagates them to the peers.
Every participating node connects to the first running process, the Head Node Process (HNP), via OOB and stores its runtime information in the General Purpose Registry (GPR), particularly its addresses. Obviously, the OOB channel also needed IPv6 porting. To prevent code duplication, the existing oob-tcp-component was extended to work with IPv4 and IPv6 connections within a single component, but on two different sockets. Though using only one socket by employing IPv4-mapped IPv6 addresses [7] would have been possible, this approach would have prevented Open MPI from running on systems where IPv6 support is enabled and IPv4-mapped IPv4 addresses are turned off, as it is commonly the case on BSD-derived systems.

The same justification holds true for OMPI’s BTL, which in fact is very similar to the OOB, but optimised for high performance message passing. The existing btl-tcp-component was extended by a dedicated socket for IPv6 connections. Unfortunately, this causes oversubscription whenever a node is reachable by IPv4 and IPv6 on the same interface. Then, message striping leads to multiple connections on the same link, causes additional overhead and finally reduced overall throughput.

To circumvent such a degradation, a selection algorithm was implemented to identify NICs and addresses on the same network. Then, Open MPI chooses the best subset of matching address pairs to maximise throughput with the help of graph theory: Local and remote interfaces denote the vertexes in a bipartite graph as shown in Figure 4, whereas edges between these disjoint sets represents possible connections. Depending on its quality, each edge is assigned a weight:

1. for private IPv4 addresses on the same IP network,
2. for public IPv4/IPv6 addresses on different networks,
3. for public IPv4/IPv6 addresses on the same network.

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

To guarantee connectivity, no edges with a weight of zero may be selected. To avoid oversubscription, no two selected edges may have a common endpoint on either side, locally or remote. A set of edges meeting this requirement is called a matching. To maximise throughput, the number of concurrent connections has to be maximised by selecting as many edges as possible. Such a set of edges is then called a maximum matching.

The algorithm solving this problem is a maximum cardinality maximum weight matching, where maximum cardinality maximises throughput by employing as many connections as possible. On the other hand, maximising weights ensures to contact only the most likely working address on the remote interface, whether it is IPv6, IPv4 public or the potentially duplicated RFC 1918 addresses in multi-cluster environments.

As of this writing, Open MPI includes fully transparent IPv6 support. Though the user can manually disable either IPv4 or IPv6 at runtime, there’s usually no need to do so, “it just works.”

4. Measurements

4.1. MPICH2

The MPICH2 measurements are done by using two nodes of a cluster at Jena running version 1.0.5p3 of MPICH2. The two machines are dual AMD Opteron 250 processors running Linux 2.6.22 connected via Broadcom BCM5704 Gigabit Ethernet adapters to a Netgear ProSafe GS724T switch. Both the IPv4 and IPv6 measurements use the Intel MPI Benchmark (IMB Version 3.0) [11].

Figure 5 shows that the throughput of the pingpong benchmarks are more or less the same. The latency of MPICH2 over IPv6 is slightly better than over IPv4 (45.9 µs vs. 44.9 µs). The maximum throughput also differs slightly (111.5 MB/s over IPv4 vs. 109.9 MB/s over IPv6).

4.2. Open MPI

IPv6 message passing delivers decent performance as shown in Figure 6. The results were also obtained by
5. Conclusion and Future Work

We have presented the IPv6 migration of the two most popular MPI-2 implementations. Since these implementations make use of an independent runtime environment, both, the network abstraction layer of the implementations and their runtime environment had to be modified.

IPv6-aware socket creation and address handling are the most important changes inside the network modules of the network abstraction layer implementations. For these parts of the implementations the porting was straightforward.

The start-up-phase of the runtime environment and the provision of network addresses to the MPI-processes required some more efforts. The initialisation of the runtime environment is the phase to obtain network addresses from the operating system to allow communication between the MPI processes. The kind of network and protocol has to be determined to choose the correct address type. The next step of the MPICH2-IPv6-implementation will be to transparently support IPv4 and IPv6 without further configuration.

Open MPI 1.3 already provides transparent IPv4 and IPv6 support, making message passing between multiple clusters work like a charm. In the future, aggregating clusters into large computational Grids might benefit even more from topology aware collectives, probably based on IPv6 multicasts, which have yet to be implemented.

Our measurements show that both MPI/IP6 implementations have similar performance compared to their IPv4 versions.

With regard to the availability of IPv6 capable MPI-2 implementations, one should consider IPv6 in addition to IPv4 for new IP-based clusters, especially in the case where not enough public IPv4 addresses are available. Limiting oneself to private IPv4 addresses only is cutting one’s options for future communication and Grid-enabled applications.

References

Figure 6. Open MPI ping-pong benchmarks for IPv4 and IPv6 show hardly any difference for (a) throughput and (b) latency.


