Survey on Architectures and Communication Libraries dedicated for High Speed Networks

Ouissem Ben Fredj*, Hassen Sallay, Mohsen Rouached, Adel Ammar, Khaled AL-Shalfan, Majdi Ben Saad

Research Unit of Sciences and Technology
Al-Imam Mohamad ibn Saud Islamic University, Saudi Arabia

Abstract
This paper studies the evolution of high performance computing (HPC) and its trends. It exposes the different architectures used in HPC, the common high-speed networks, the programming models, the communications models, and the communication libraries.

Keywords: HPC, Architecture, Cluster, Communication layer, High Speed Network

1. Introduction
From the beginning of computer sciences, HPC (High Performance Computing) has followed big changes in order to handle the growing performance needed by organizations. These changes include the hardware architectures and components, the network architectures, and the software. The introduction of high-speed networks have enhanced the use of specific architectures and programming models. This paper studies the evolution of HPC and depicts the most impressive components and its trends. Next section focuses on the history and the trends of HPC architectures. Section 3 is a survey of most used high-speed networks. Section 4 studies the different communication libraries, its programming models, and its communication models.

2. HPC Architectures: history and trends
The evolution of HPC began in the late 50s. At the end of the 60s, some techniques like multithreading, time sharing, and virtual memory became common. Research in parallel computing became mature and the skeleton of main machine architectures was established. The most well-known classification for parallel architectures was introduced by Flynn [9] in 1966 where four classes (SISD, MISD, SIMD, MIMD) were defined based upon the nature of the instruction (or control) flow and the nature of the data streams available in the architecture. The discovery of MOS (Metal Oxide Semi-conductor) in 1971 allowed a quick and reliable processor at an acceptable price. Thus, HPC entered the industrial era.

2.1. The Industrial Era
Most companies have implemented the SIMD architecture, and especially vector supercomputers. The limitations of the SIMD model have led manufacturers to build another kind of architecture which is easier to program and fits with a larger class of applications. The MIMD model with its two variants shared memory and distributed memory was the answer to these limitations. Designed at the beginning of 60s, MIMD was the model of most of parallel machines in the late 80s and in the 90s [1]. The introduction of the message-passing programming paradigm for distributed memory systems, in particular MPI, enhanced the commercialization of MIMD-DM machines. On the other hand, new customers were not mainly interested in performance, but also in system reliability, continuity of the manufacture, fast update, software support, flexibility, scalability of the architecture, application code portability, acceptable prices and a global view for IDSs, firewalls, and proxies. For these aims, researchers invented a new generic architecture based on general-purpose computers with a complete local memory and a scalable network called cluster.

2.2. The Cluster
Depending on the level of parallelism in term of the number of nodes and processors in each node, TOP500 distinguishes two classes of clusters: cluster-NOW (more nodes than processors) and constellation (more processors than nodes) which both are commodity cluster systems. This distinction has a serious impact of the way the cluster is programmed. In June 2010, 424 cluster-NOWs and 2 constellations were in the TOP500 list [1]. They together represent 85.2% of the most powerful parallel machines in the world. As predicted by different researchers [8], clusters will have a long lifetime as an architecture principle. This is because one can always build a new parallel machine more powerful than the existing one through clustering. For example, the network management and control which include IDSs, firewalls, proxies, and authentication points, need on the one hand a global view of a scalable network and on the other hand a high-performance communication and computing systems to avoid the bottleneck. These two requirements seem to be a paradox with centralized architecture. But, it becomes possible with clusters!
2.3. Cluster of Clusters

Researches in Grid Computing have suggested different solutions to use a set of clusters simultaneously which leads to the new parallel architecture known as Cluster-of-Clusters (CoC) or MultiCluster. This new computational environment may incorporate different clusters with different network types, processor types, and memory sizes. In addition, a CoC potentially integrates several heterogeneous independent operating systems and has a highly heterogeneous and unbalanced communication network, comprising a mix of different intra-cluster networks and a variety of inter-cluster connections whose bandwidth and latency may vary greatly. Several grids and Cluster-of-Clusters platforms have been set up. Some representative research platforms include Grid’5000 [6] in France, NAREGI, the Hungarian SuperComputing Grid, and DAS-3 (The Distributed ASCI Supercomputer 3) which are more suitable to run experiments. Production platforms include Grid over the GARR, Clusterix, and Cluster Grid [15] which provides a good quality of service.

The common properties shared between clusters and Cluster-Of-Clusters is the existence of high-speed network links within clusters. High-speed networks have been the key for performance in clusters.

3. High Speed Networks

Since clusters comprise commodity nodes, interconnection systems (both hardware and software) that connect those nodes have become the main responsible for cluster performance. In the following, architecture choices of most used SANs is described: Gigabit Ethernet, InfiniBand, and Myrinet. 89.80% of the 500 most powerful commercially available computer are based on those SANs (in the top500 list of June 2011) [1].

3.1. The Gigabit-Ethernet

According to the top500 list of June 2010, 48.4% of the 500 most powerful computers in the world use Gigabit Ethernet as the network interconnect. Gigabit Ethernet, known as IEEE Standard 802.3z, offers a one gigabit per second (1 Gbps) raw bandwidth. Gigabit Ethernet is most effective when running in the full-duplex, point-to-point mode where full bandwidth is dedicated between the two end-nodes. Ethernet have followed big improvement in the beginning of the century. In 2002, a 10-gigabit Ethernet standard was published. Then, in 2007, a 100-gigabit Ethernet standard [17] was proposed, and authorized in June 2010. The idea behind that is to send frames over multiple 10-gigabit Ethernet lanes.

3.2. Myrinet 2000 Technology

Developed by Myricom Inc., Myrinet-2000 [19] is a proprietary network technology and is compliant to the Physical and Data Link layer defined in the ANSI/VITA 26-1998 standard. Myrinet is a switched, Gigabit per second network that is widely used in clusters. A Myrinet-2000 network is composed of crossbar switches and network adaptors.

Network adapters are connected to the switches through point-to-point duplex links and crossbar switches can be interconnected in an arbitrary topology. The basic building block for Myrinet-2000 switches is a 16-port crossbar (XBar16). The topology used to interconnect these switches is a full-bisection Clos network.

By applying the same Clos network principle, Myrinet-2000 can scale up to 8192 hosts that can potentially offers a network bisection bandwidth in the order of Terabits per second. Data packets are wormhole routed from one network adapter to another through a series of crossbar switches enabling a latency of approximately half a microsecond. Flow control is achieved by inserting STOP and GO control bytes into the opposite channel of a link by the receiver side to stop or restart data transmission on the sender side. As a result, Myrinet would normally not drop packets unless the receiver fails to drain the network. Error control is accomplished by computing a cyclic-redundancy check (CRC) on the entire packet including packet header. The CRC is then carried in the packet trailer and recomputed in each network stages. Thus, data packet entering a host interface with a non-zero CRC indicates transmission errors.

Myrinet adapters have a programmable network interface processor known as LANai. Several LANai versions exist, the most recent are LANai9, LANaiXP, LANai2XP, and LANaiXM. LANaiX versions allow pipelining for all receive, send and host-LANai DMA operations, in addition to two hardware components for CRC32 computation and next-event-to-service computation. Our testbed is equipped with LANai9 chips. LANai9 is a 32-bit RISC processor that operates at up to 133 MHz for PCI64B interfaces, or at up to 200 MHz for PCI64C interfaces. Using the Myrinet Control Program (MCP), which is stored in the onboard static RAM (SRAM), LANai9 controls the data transfer between the host and the network, performs data buffer management (through memory interface), and maintains network mapping and monitoring. The benefit of a programmable network processor is that it enables researchers to explore many protocol design options.

The onboard Myrinet-2000 SRAM size is less than 8 MB and operates at the same clock speed as LANai9. Thus the maximum attainable bandwidth is approximately 1,064 Mbytes and 1,600 Mbps for the PCI64B and PCI64C respectively. The SRAM is accessible from both the onboard local bus (LBUS) and the external system bus (EBUS). LBUS and EBUS are both 64-bit wide but the LBUS is clocked at twice the chip-clock speed, permitting two LBUS memory accesses per clock cycle.

To increase the data transfer rate, a Myrinet-2000 network adapter is equipped with three DMA engines. Two DMA engines are associated with the packet interface: one for receiving packets and one for sending packets. The third DMA engine is used for data transfer between the SRAM and the host system memory through the host interface. Like most systems that support DMA, the onboard memory can be mapped into user space and is thus accessible directly to user processes. In order to support zero-copy APIs efficiently, the DMA operations can be performed with arbitrary length and byte alignments.

Additionally, the DMA engines also compute the IP checksum for each transfer and provide a "doorbell" signaling mechanism that allows the host to write anywhere within the doorbell region, and have the address and data stored in a FIFO queue in the local memory.

The host processor can also access the Myrinet-2000 SRAM through the programmable input/output (PIO) interfaces. With PIO, the host processor reads the data from the host memory and writes it into the Myrinet-2000 SRAM. This mode of data transfer typically results in many PCI I/O bus transactions. Although Myrinet-2000 PCI64 interfaces are capable of sustained PCI data rates approaching the limits of the PCI bus, the network performance greatly depends on the data transfer rate of the host's memory and the PCI-bus implementation. PIO transactions can be combined to perform one Write-Combining transaction with only one PIO startup.
Myricom recommends the GM API for LANai9 and LANaiX and the MX API for LANaiX and LANai2X. Although, since the LANat is programmable, many message passing libraries (RWAPI, AM, FM, BIP, MyVIA...) provide their own MCP implementing a specific network protocol. Note that recent LANatX versions require a significant work to be programmed. Other efficient middleware implementations over Myrinet are available from Myricom and from third parties.

3.2. InfiniBand

The InfiniBand Architecture (IBA) [12] is an industry-standard architecture for server I/O and inter-server communications. InfiniBand is an I/O channel-based architecture [16] rather than register based. A channel-based architecture off-loads the inter-processor communication overhead from the CPU to provide maximum available performance. In an InfiniBand based system, I/O operations are scheduled as queued DMA operations and are handled by the node hardware rather than the CPU.

The primary architectural element is the Queue Pair (QP). Each QP consists in an outbound queue, and an inbound queue. Queue Pairs are not shared between applications; therefore, once they are set up, they can be managed at the application level without incurring the overhead of system calls. The QP is the mechanism by which quality of service, system protection, error detection and response, and allowable services are defined. An application can use many QPs, each one with a different quality of service. Creating a QP requires support from the operating system which handles the HCA and initializes memory regions to be used by QPs to manage communication requests and memory operations. The available transport service types include: Reliable Connection, Unreliable Connection, Reliable Datagram, Unreliable Datagram and Raw.

Reliable Connection (RC) provides the highest level of reliability and predictability. Unreliable Connection (UC) also consists in a dedicated channel between local and remote QPs. However, in this case, there is no acknowledgment provided but the operations complete in order. Unreliable Datagram (UD) is a connectionless and unreliable service. Reliable Datagram (RD) combines the features of both RC and UD services. Essentially, it provides a multiplexed reliable connection channel. Operations are acknowledged, complete exactly once, complete in order, and are automatically retried on error. And finally, Raw is used to send and receive packets for a protocol other than InfiniBand such as IPv6 or Ethernet packets. It supports only the Send operation.

4. Communication Libraries

Several parallel communication libraries have been developed in order to use efficiently the features of the host and the network hardware introduced in the last two sections. They mainly differ in the programming model, the communication model, the software design, the way the libraries communicate with the OS and the NIC, and the network interconnects they support. This section describes the most representative communication libraries used with high-speed network in cluster environments.

4.1. VAPI

The architecture of InfiniBand is message passing. It incorporates many of the concepts of the Virtual Interface Architecture [3]. Each vendor has a different software stack with a proprietary value-add. However, there are multiple vendor-independent access layers that support different HCA (Host Channel Adapter, the network card) simultaneously called verbs. The Mellanox IB-Verbs API (VAPI) interface provides a set of operations that are close to the verbs of the InfiniBand standard, plus additional extension functionalities in the areas of enhanced memory management and adapter properties specifications. The VIEO InfiniBand Channel Abstraction Layer (CAL) Application Programming Interface provides a vendor-independent interface for InfiniBand channel adapter hardware. The CAL API evidently lies under the verbs abstraction, between the Channel Interface driver software and adapter hardware. It isolates specific hardware implementation details, providing both a common function call interface and a common data structure interface for the supported InfiniBand chipsets. An advantage of such an abstraction is the simultaneous support for heterogeneous channel adapters. This potentially enhances path selection.

In addition to the above interfaces that can be used to communicate with the Host Channel Adapters directly, there also exist some portable interfaces that hide the channel access interface from the user (see Fig. 1). These include the SRP, the IPoIB, the SDP, and the SM. The SRP (SCSI RDMA Protocol) enables access to remote storage devices across an InfiniBand fabric. The SM (Subnet Manager) handles several areas related to the operation of the subnet including: discovery, monitoring and configuration of the ports connected to the subnet, responding to subnet administrative (SA) queries, configuration of I/O units with host channel drivers, performance management, and baseboard management.

The IPoIB provides standardized IP encapsulation over InfiniBand fabrics as defined by the IETF. This driver operates only within an InfiniBand fabric. The SDP (Sockets Direct Protocol) is an InfiniBand specific protocol defined by the Software Working Group (SWG) of the InfiniBand Trade Association (IBTA). It defines a standard wire protocol over IBA fabric to support user-mode stream sockets (SOCK_STREAM) networking over IBA. The SDP is a good alternative to guarantee the co-existence of socket streams and InfiniBand packet without major performance degradation.

There are many implementations of these components resulting in many software stacks. These stacks include the OpenIB which provides an open source implementation dedicated to the Linux 2.6 kernel, the IBAL (Intel Sourceforge InfiniBand project) which is the Intel open-sourced software stack that is adapted to Linux 2.4 kernels, the IBGD (IB Gold Distribution) stack which is a full InfiniBand upper layer protocol stack for the Linux operating system hosted by Mellanox.

![Fig. 1: The Mellanox InfiniBand-Verbs API (VAPI) software stack.](image)
4.2 Myrinet Express

Myrinet Express (MX) [20,21] is a low-level message-passing software interface for Myrinet interconnects. The MX API is close to MPI and thus provides a flexible interface for modern middleware such as MPI or VIA. It also enables Ethernet emulation. The MX API includes a protected and independent access for user-level applications. MX endpoints virtualize the network interface at the process level, providing OS-bypass communications. It supports a large number of independent endpoints and provides a native support in the NIC for several endpoints. The matching protocol is ordered along endpoints. Despite, MX does not ensure in-order delivery in independent endpoints and provides a native support in the NIC for several endpoints. The matching protocol is ordered along endpoints. Despite, MX does not ensure in-order delivery or notification. Out-of-order delivery and notification is possible since MX supports route dispersion and multiple ports per NIC. This ensures reliability for a large number of out-of-order packets without requiring retransmission. However, when the number of out-of-order packets exceeds available buffering resources, MX drops any other out-of-order packets.

MX provides a fully asynchronous communication primitive. Once an operation has been initiated, the application is not involved until it checks or waits for it, thus allowing an overlap between communication and computation. Collective operations are also fully asynchronous. While the number of pending sends and receives natively supported by MX at the NIC level is large, MX offers a multiplexing capability to provide an unlimited number of pending sends and receives. MX distinguishes between four types of message: tiny, small, medium, and large. MX encapsulates tiny message in the send descriptor which will copied in the NIC memory using PIO. At the receive side, the data is copied using one DMA request in the event queue. Finally, MX copies the data to the application's memory space. For small message, the data is copied to the NIC memory using a dedicated PIO request, thus resulting on two PIO requests in the sender side. At the receiver side, the data is copied to the receive queue using one DMA request in addition to the DMA request for the receive event. Then, the data is copied to the application's memory space.

A medium message transfer is processed in the same way as a small message. The only exception is that the sender copies the data to a pre-pinned area in the MX memory space in order to call the DMA engine to copy it to the NIC memory. With large message, MX tries to apply DMA both at the sender and the receiver sides. MX sends a signal request to check if the receiver process has posted the correspondent receive request. If it is the case, the receiver sends safely the data using DMA both at the sender side and the receiver one after it posted a send request using PIO. Otherwise, the receive process performs a mx_get() request as soon as a receive request is posted, indeed using DMA on both sides. Note that the MX driver is requested to perform the translation from/to virtual addresses to/from physical addresses.

Regarding communication control, MX provides functions to check the completion of a specific pending operation or the completion of all of the pending operations related to an endpoint. Similarly, there are functions to block waiting for completion of a specific pending operation or the completion of all of the pending operations related to an endpoint. These blocking semantics release the processor for other application computations.

In order to work around the synchronization problem of the rendez-vous model, MX introduces unexpected messages. An unexpected message is one for which a matching receive request has not been posted yet. Unexpected messages are processed by receiving the entire message eagerly in an unexpected queue if it is small, and by receiving only its header if it is too large. The threshold distinguishing the handling method can be controlled by the application. MX guarantees in-order matching even if unexpected messages have been buffered.

MX tries to recover from faults by retransmitting packets or routing around dead links. However, interrupt is used for catastrophic or unrecoverable errors due to hardware or software failure. MX provides both thread-safe and single-threaded libraries to allow users to select which is the most appropriate for the application.

4.3. MPICH2

MPICH[11] was developed at Argonne National Laboratory in 1993 in an effort to provide an implementation that would closely track the MPI standard definition as it evolved. It has been ported to many different systems, including: clusters running different Unix/Linux variants; Windows NT and Windows 2000 networks; MPPs such as Thinking Machines CM-5, IBM SP, and Intel Paragon, and SMPs. As a successor of MPICH, MPICH2 [14] aims at supporting not only the MPI-1 standard, but also functionalities such as dynamic process management, one-sided communications and MPI I/O, which are specified in the MPI-2 standard. However, MPICH2 is not merely MPICH with MPI-2 extensions. It is based on a completely new design, aiming at providing more performance, flexibility and portability than the original MPICH. One of the notable features in the implementation of MPICH2 is that it can take advantage of RDMA operations if they are provided by the underlying interconnect. These operations can be used not only to support MPI-2 one-sided communications, but also to implement normal MPI-1 communications.

The major components of the MPICH2 software structure (see Fig. 2) are the PMI (Process Manager Interface), ADIO (Abstract-Device Interface for I/O) and the ADI3 (the third generation of the Abstract Device Interface). PMI provides a scalable interface to both process creation and communication setup. It is designed to permit many implementations, including with/without daemons and third-party process managers. ADIO is a mechanism designed for implementing parallel-I/O APIs on multiple file systems.
Channel interface contains only five functions, among which only two are central to communications. (Other functions deal with process management, initialization and finalization). These two functions are called put (for RDMA-write) and get (for RDMA-read).

Several implementations of MPICH2 over different platforms exist. Examples include IBM’s MPI for the BG/L, Cray’s MPI for the Red Storm and XT-3, Microsoft MPI, Intel MPI, MVAPICH2 for InfiniBand, MPICH2 over SCTP (SCPT), which is a message-oriented reliable transport protocol recently standardized by the IETF and designed to bridge the gap between UDP and TCP.), MPICH2 for ATOLL interconnect, Chemnitz InfiniBand, DeinoMPI for Windows. However, MPICH does not currently support these implementations at the same time as they are independent and they cannot cooperate to perform communications over heterogeneous environments.

4.4. Madeleine

Madeleine III [29] is a distributed communication library for clusters and Cluster-of-Clusters. The Madeleine interface is message-passing oriented and provides an incremental message management policy. The low layer of Madeleine is based on two major notions: the connection and the channel. The connection is an abstraction of a one-way point-to-point communication link between two nodes. Messages sent over a connection are FIFO-ordered. The channel object is an abstraction of a cluster (or a network as called in the Madeleine design) characterized by a name, a set of nodes and a network. According to Madeleine, a cluster may be a physical cluster or a virtual cluster. Virtual clusters are based on physical ones and are mostly used to create a Cluster-of-Clusters. For instance, if a node belongs to both channels virtualizing a SCI cluster and a Myrinet cluster, this node will play the role of a gateway interface launchers. However, Leonie supposes a one-level tree-based topology, thus it could not spawn processes on a private node of a remote cluster.

The implementation of the ADI of MPICH on top of the version II of Madeleine called ch_mad [30] performs a seamless communication between heterogeneous nodes over different network types. The ch_mad layer supports three transfer modes: the short mode for message less than 16 bytes, the eager mode and the rendez-vous mode.

4.5. MPICH-G2

MPICH-G2 [13] has been developed at Northern Illinois University and aims at taking advantage of the Globus Toolkit v2 (GT2) [10] to provide a grid-enabled MPI implementation. MPICH-G2 implements the MPI-1 standard and several functions of the MPI-2 standard, in particular the client/server management functions. It makes use of GT2 to provide another implementation of the ADI layer of MPICH which is also called the globus2 device. GT2 is mainly used to hide the heterogeneity during startup and management.

Fig. 3 shows the different modules of MPICH-G2. MPICH-G2 requires Grid Security Infrastructure (GSI) [5] to authenticate the user to each site. The user must provide Resource Specification Language (RSL) [7] scripts that identify the topology resources and specify job’s requirements and parameters for each site of the topology. Once authenticated, the user calls the standard mpirun command to start the MPI application. Then, MPICH-G2 calls the Dynamically-Updated Request Online Coallocator (DUROC) to schedule and start the application.

Communications in MPICH-G2 are split into two types: the vendor-supplied MPI (vMPI) if available, and Globus communications (Globus IO) with Globus Data Conversion (GlobusDC) using TCP. TCP communications are split further into wide area, local area, and system area communications. Wide area communications are those between two different sites; local area communications include those between two different machines at the same site; and system area communications are those between different compute nodes of a cluster or even between different processes on the same machines. Using these topology depths, MPICH-G2 could group processes at a particular level by assigning different colors. Two processes having the same color at a particular level can communicate at the network level. Topology depths and colors are used to optimize communications.

![Fig. 3: MPICH-G2 components involved at startup.](image)
4.6. LAM

LAM (Local Area Multicomputer) [4] was originally developed at the Ohio Supercomputing Center in 1989 and moved to Indiana University in the fall of 2001. LAM supports all of MPI-1.2 standard and much of the MPI-2, including dynamic process creation and management.

LAM has been tested on clusters and network of workstations running various UNIX and Unix-like operating systems, including Solaris, OpenBSD, Linux, Mac OS X, IRIX, HP-UX, and AIX. As dedicated clusters have gained popularity, LAM has also gained popularity because it was originally developed for the cluster architecture.

![Component architecture for LAM](image)

Fig. 4 shows the component architecture of LAM. Each component is a set of small independent modules. Hence, it is possible for multiple modules of the same component type to co-exist in a process so as to be selected for use at run-time.

Basically, LAM is made of the runtime environment (RTE) and the MPI communication layer. For performance reasons, both layers interact directly with the operating system. RTE is based on user-level daemons (called lamds), and provides message-passing services, process control, remote file access, and I/O forwarding [4]. The boot component provides services for launching the LAM RTE. It can use rsh/ssh, TM (OpenPBS / PBS Pro), SLURM, BProc, or Globus to remotely start the daemons. Although, the time for lamboot to execute can be long on large platforms using rsh/ssh.

The MPI layer supports three major types of components. RPI (Request Progression Interface) components provide the back-end implementation of MPI point-to-point communication. The Coll components provide the back-end implementations of MPI collective algorithms. CR components provide interfaces to checkpoint-restart systems to allow parallel MPI jobs to be checkpointed.

Component-based RPI provides the capability to support different underlying message-passing transports in a single installation of LAM. Several modules of RPI exists, each implemented on top of different network interconnect communication libraries. This includes TCP, two shared memory communication channels, each using TCP for remote-node communication, Myrinet networks using the GM interface, Infiniband networks using VAPI.

LAM implements much of the Interoperable MPI (IMPI) standard, intended to allow an MPI application to execute over multiple MPI implementations. The use of IMPI allows users to obtain best performance, even in a heterogeneous environment.

4.7. Open MPI

Open MPI merges between three well-known MPI implementations: FT-MPI [18] from the University of Tennessee, LA-MPI [22] from Los Alamos National Laboratory, LAM/MPI from Indiana University, and PACX-MPI [23] from the University of Stuttgart. Open MPI [24] is not yet another MPI implementation but it represents the next generation of each of these implementations. The software implements both MPI-1.2 and MPI-2 specifications and guarantees a concurrent, multi-threaded execution. Open MPI natively supports a wide range of networks for point-to-point communications. These include TCP/Ethernet, Shared memory Loopback (send-to-self), Myrinet (GM and MX), Infiniband (OpenIB and VAPI), and Portals.

The primary software design element of Open MPI [24] is the Modular Component Architecture (MCA). MCA manages different Component frameworks which in turn manages different Components. Open MPI has three classes of components: Open MPI components (OMPI), Open Run-Time Environment (ORTE) components, and Open Portable Access Layer (OPAL) components.

![Component architecture for Open MPI](image)

ORTE provides a uniform parallel run-time interface regardless of the system capabilities. Four Component frameworks belongs to this class: GPR for general purpose registry, IOF for I/O forwarding, RDS which is the resource discovery system, and PLS for process launch system that uses several tools including rsh, XCPU [25], XGrid [26], BProc [27], fork, POE, SLURM [28], and TM. OPAL abstracts the particularities of a specific system away to provide maximum portability, in particular the memory and the processor elements of the system.

The most important OMPI component frameworks are:

- The Point-to-point Management Layer (PML) which implements the semantics of a given point-to-point communication protocol such as MPI;
- The Byte-Transfer-Layer Layer (BTL) which handles point-to-point data delivery over the network, and is unaware of upper-level point-to-point communication protocols such as MPI;
- The BTL Management Layer (BML) that provides services during job startup and dynamic process creation to discover and maintain the set of BTLs that may be used for point-to-point communications between a given pair of endpoints;
- Collective Communication (COLL) which is the back-end of MPI collective operations, supporting both intra- and inter-communicator functionalities;
- The Process Topology (TOPO) that uses cartesian and graph mapping functionalities for intra-communicators. Cluster-based and Grid-based computing may benefit from topology-aware communicators, allowing MPI to optimize communications based on locality;
- MPool which provides memory registration and deregistration services for DMA based operations and used by both PML and BTL components;
- Reache provides facilities for caching and searching for DMA registrations.

Open MPI supports two forms of network heterogeneity. It can stripe a single message to a single destination over multiple networks (of either the same or different communication protocols) and it can communicate to different peers using different communication protocols. In order to add an additional network support, only one instance of BTL, MPool and Reache components has to be added. Fig. 5 shows the relation between the different components. PTLs are controlled by five major stages of action: opening, initializing, communicating, finalizing, and closing.

A PTL component goes through the first two stages: opening and initializing, to join the communication stack. A PTL component is to be opened as a shared library. When it is loaded successfully, a process initializes the network interfaces, prepares memory and computing resources, and fills in the correct fields of PTL modules (one per network interface).

PML schedules messages across a new network when the PTL component has activated its PTL modules. When the PML layer receives a request, it schedules the first packet to one PTL based on a chosen scheduling heuristic. For large messages, this packet serves as a rendez-vous packet to the receiver. When it is received by one of the PTLs, the receiving PTL asks the PML layer to match this packet to the pre-posted receive requests. An acknowledgment is returned to the initiating PTL if this is a rendez-vous packet. Any data inline with the first packet are copied into the application receive buffer. When the acknowledgment arrives at the sender side, the initiating PTL updates the PML layer about the amount of transmitted data. Another scheduling heuristic is then invoked to schedule the rest of the message across available PTLs. During the finalizing stage, a PTL first finalizes its pending communications with the other peer processes, then releases all the associated memory and computing resources. The open shared library is closed after all the exposed PTL modules are finalized.

4.8. RWAPI: Remote-Write Application Programming Interface

Remote Write [2] is a simple communication protocol in which the sender of a message is in charge of providing all the information needed to copy a contiguous memory area from one node to another. RWAPI distinguishes between small messages used for control and large message used for data transfer.

Fig. 6 shows the different steps involved in a large message transfer using the Remote-Write API:

1. The application writes the content of the message in a previously allocated contiguous memory area.
2. The application adds a new entry in the send queue. All information required to perform the message transfer are provided: the destination node number, the local address, the remote address (where the message will be copied on the receiver) and the size of the message.
3. The application informs the NIC that a new entry has been posted in the send queue.
4. The NIC reads the next available entry in the send queue.
5. Using the local address and the size of the message, the NIC reads the message content from memory.
6. and sends it to the destination node using the network.
7. When the message arrives on the destination node, it is copied in memory using the remote address and the size of the message.
8. After the message has been received completely, the NIC adds a new entry in the receive event queue.
9. The application is then able to read the content of the receive event queue to take into account new incoming messages.
10. When a message is handled, the application notifies the NIC to free the receive event queue.

5. Conclusion

This paper presented the history of high-performance computing. Along the presentation of the different phases of the evolution, we noticed a clear convergence to a generic parallel architecture which is cheap, extendable, and which performance can be compared to those of supercomputers called cluster. The common property shared between clusters is the existence of high-speed network links between nodes. High-speed networks have been the key for performance in clusters.

Three high-speed networks were studied: gigabit Ethernet, Myrinet, and Infiniband. Each of them has strengths and weaknesses. Among the shared properties of the above SANs, first is the distinction between short messages and large messages. Each network type suggests a different way to deal with these two types, but both aims at providing a low latency for short messages and a high bandwidth for large messages. We notice a clear convergence to the rendez-vous model which requires that a receive request must be posted before the corresponding send request. They try to work around the problem when no receive request is provided which leads to an expensive wait at the receiver side or to extra-copies while buffering. This model allows the use of new hardware features such as DMA and RDMA.

On other hand, many works have been done to take advantage of the cluster architecture. The section four presented some popular parallel communication libraries which aim at providing a simple user interface while hiding complicated low-level layers. The most used programming model is message-passing based on the rendez-vous mode. This mode suffers from a synchronization problem. Several challenges have been faced to these libraries. Especially when the libraries support several processor architectures and several network types. The major contribution of these libraries was the introduction of the component-based or the object-oriented-based software architecture which enable a runtime instantiation of different modules. Their portability advantage usually leads to a degrading of performance results.
References