

Chapter 8

SCore



**Atsushi Hori, Hiroshi Tezuka, Shinji Sumimoto, Toshiyuki Takahashi,
Hiroshi Harada, Mitsuhsa Sato and Yutaka Ishikawa**

Abstract SCore is a software package for high- performance clusters. It includes a low-level communication layer named PM(v2), a user-level, global operating system called SCore-D, an MPI implementation, an OpenMP compiler that enables OpenMP programs to run on distributed memory clusters, as well as other cluster management utility programs. SCore was developed by the Real World Computing Partnership project during 1992–2002. SCore provided state-of-the-art technologies at the time. Some of the technologies became obsolete but some of them, e.g., gang scheduling and checkpoint with parity are still unique.

A. Hori (✉) · H. Harada · M. Sato · Y. Ishikawa
Riken R-CCS, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan
e-mail: ahori@riken.jp

H. Harada
e-mail: hiroshi.harada@riken.jp

M. Sato
e-mail: msato@riken.jp

Y. Ishikawa
e-mail: yutaka.ishikawa@riken.jp

H. Tezuka
Department of Creative Informatics Graduate School of Information Science and Technology,
The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
e-mail: tezuka.hiroshi@ci.i.u-tokyo.ac.jp

S. Sumimoto
Fujitsu Ltd., 4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki 211-8588, Japan
e-mail: sumimoto.shinji@jp.fujitsu.com

T. Takahashi
METAHACK Japan, K.K., 2-4-4-201 Honkugenuma, Fujisawa, Kanagawa 251-0028, Japan
e-mail: toshi@metahack.jp

8.1 Background

SCore (pronounced as [*es-core*]) is the name of a software package for high-performance clusters. It includes a middleware global OS, a low-level communication library, an MPI implementation, etc. SCore was developed in the Real World Computing Partnership (RWCP) project funded by Japanese government from 1992 to 2002, a 10-year project. Shortly after the beginning of the project, a variety of user-level communication libraries were developed, e.g., U-Net (von Eicken et al. 1995), AM (von Eicken et al. 1992), FM (Pakin et al. 1997). The Myrinet network (Boden et al. 1995), which had a programmable processor in the network interface card (NIC), played a big role in these developments. The SCore development team was no exception, so the PM(v2) communication library on Myrinet had been developed (Tezuka et al. 1997). Unlike other user-level communication libraries, however, the PM(v2) was designed with OS support in mind.

Another dominant trend of the time was the increasing prevalence of Beowulf clusters (Sterling et al. 1995). Thomas Sterling et al., developed a parallel machine, the so-called *Beowulf Cluster*, combining commodity parts and software, e.g., x86-based PCs, Ethernet, and Linux. As opposed to the Beowulf clusters, the SCore software package aimed at higher performance clusters.

In the early 90s, clusters were called COWs (Cluster of Workstations) or NOWs (Network of Workstations). SCore was first developed on SunOS (BSD) as the global OS for COWs or NOWs (Fig. 8.1), then moved to NetBSD. Finally, SCore was ported to Linux.



Fig. 8.1 Workstation cluster of 36 Sun workstations connected with Myrinet installed at RWCP, 1996. This cluster was used to develop the first version of SCore

8.2 Technical Details

8.2.1 SCore Overview

SCore is the name of a software package for HPC clusters. SCore-D is the global operating system to manage clusters (Hori et al. 1998). Since SCore-D runs on top of Unix (SunOS, NetBSD or Linux) on each node, it can be considered a middleware operating system. The SCore software package also includes the *scout* parallel shell environment, *PM(v2)*, a low-level communication library for Myrinet, Ethernet, InfiniBand and Shmem for intra-node communication, *MPC++*, (a multi-threaded C++ for distributed memory machines), an MPI implementation utilizing *PM(v2)*, *SCASH*, a software-distributed shared memory implementation, and *OMNI*, an OpenMP compiler on top of *SCASH*. All of these programs and libraries were developed in the RWCP project. Most of them were unique and state of the art at the time.

8.2.2 SCore-D and PM(v2)

PM was originally developed as a communication library for Myrinet. Later, it was enhanced to support multiple protocols including Ethernet and Shmem, and it was renamed to *PM(v2)*. *PM(v2)* supports zero-copy communication and proposed the pin-down cache (Tezuka et al. 1998), which has been widely used in other communication libraries.

We believed that for ease of use there was a need for time-sharing scheduling that gives users shorter response times than that of batch scheduling. At the time of the SCore development, the RWCP project obtained a CM-5 machine which supported time-sharing scheduling (i.e., gang scheduling). Another part of our motivation came from experiences of using Cray-1, which also switched from batch scheduling to time-sharing.

SCore-D, as an OS, had to guarantee the soundness of user jobs and *PM* had to provide a mechanism for that. *PM(v2)*'s endpoints were created by SCore-D and then passed to user processes. The routing table of *PM* was set by SCore-D to prevent user processes from sending messages to nodes outside of the set allocated to a particular job. The functions needed for the gang scheduling were privileged and only accessible from SCore-D. Thus, each endpoint had two ports, a privileged one for SCore-D and another one for user processes.

Among various user-level communication libraries, *PM(v2)* was the only one that supported these global OS functionalities. However, these ideas did not widely spread and SCore would face problems with the forthcoming InfiniBand NIC (Fig. 8.2).

Local Scheduling

Unix operating systems (including Linux) support time-sharing scheduling. Each process is assigned a time slot, often several 10 milliseconds, which renders the

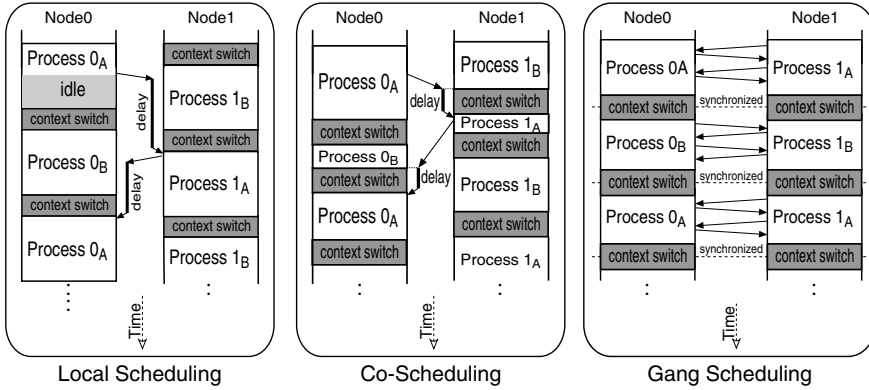


Fig. 8.2 Comparison of local scheduling, co-scheduling, and gang scheduling. “Process NX ” means it is the N -th process of job X . Process 0_A and 1_A , or 0_B and 1_B communicate with one another in the same job

process eligible to run for the given period, unless it blocks on a system call. When the time slot is exhausted or the process is blocked for some reason, the OS scheduler selects another process to run and stops the execution of the current process, i.e., it switches the context and resumes execution of the new process.

This time-sharing scheduling of Unix (or Linux) works well on sequential processes. However, usually a parallel job consists of multiple processes running on different nodes. This means that context switches on different nodes may happen independently, not in a synchronized fashion. Let us consider the case where process 0 of a job sends a message to process 1 of the same job running on different nodes and process 0 waits for the answer of the message. If process 1 is not scheduled, then process 0 must wait until process 1 is scheduled and sends back a reply. Thus, waiting time of process 0 is wasted. This asynchronous scheduling in a cluster (i.e., over a set of compute nodes) is called *local scheduling*. To reduce waste, schedulers running on cluster nodes must be synchronized in some way.

Co-scheduling

When a process waits for an incoming message, the OS can block its execution and switch to another process, eligible to run. When a message arrives to one of the descheduled processes, which is waiting for incoming message(s), then the process becomes eligible to run. All processes eligible to run in a cluster are processes not waiting for incoming messages. This scheduling method is called *co-scheduling*. Co-scheduling can hide the communication latency by scheduling other processes.

Note that the communication latency must be larger than the time needed for context switching. Otherwise the communication latency cannot be hidden and the context switch overhead can hinder the execution of a parallel job. In general, this co-scheduling is effective when communication happens infrequently and communication latency is high. However, most HPC applications communicate very fre-

quently and the latency of high-performance networks is low. Another problem of co-scheduling is that a process must wait for a message by a blocking system call and kernel drivers usually control the NIC by utilizing interrupts. However, user-level communication aims at eliminating the overhead of system calls and interrupt handling to achieve high communication performance. Thus, there is a disconnect between concepts of co-scheduling and user-level communication.

Gang Scheduling

Gang scheduling is a time-sharing scheduling technique where all processes of a parallel job are scheduled in a fully synchronized way. Let us assume that a job consists of process 0 and 1, running on different nodes. Gang scheduling synchronizes execution and all processes of a job are either in running or blocking states. There is no additional latency but the context switch overhead of gang scheduling. Thus, gang scheduling and user-level communication are well suited to each other.

Which one is the best approach?

Uncoordinated local scheduling results in low performance of job execution. In co-scheduling, processes are naturally coordinated by message reception, but high overhead, system-level communication must be applied. Gang scheduling may suffer from higher overhead because of global synchronization. Which one is better between co-scheduling and gang scheduling? The answer depends on many system parameters, e.g., the communication latency (system-level communication in co-scheduling, user-level communication in gang scheduling), process context switch duration, the overhead of global co-ordination in gang scheduling, etc. Co-scheduling works well if coordination by messaging does not depend on the communication pattern of the job. In contrast, the overhead of gang scheduling does not depend on the applications' communication pattern.

8.2.3 Network Preemption

To enable efficient time-sharing, our parallel and global operating system, named SCore-D, and the low-level communication layer, first called PM and later renamed to PM(v2), were codesigned from the beginning. The most important technical issue here was the handling of inflight messages. When a job is scheduled, all inflight messages at the time of the previous context switch must be restored. If a network is a direct network, i.e., where NICs are attached to a router and there are no network switches, it is possible for the CPU to save and restore inflight messages in the NIC and instruct the router when to switch jobs. If a network is indirect, i.e., where there are other switches or router boxes apart from compute nodes, it is very hard for the CPU to access the switch status. CM-5, which was the first commercial and parallel machine supporting practical gang scheduling, had an indirect network and inflight messages were rerouted to go to the nearest nodes at the time of context switching (Leiserson et al. 1996). PM was first developed on Myrinet which is categorized as an indirect network, and had no support for saving or restoring switch status.

Myrinet and *Network Preemption*

Myrinet network interface card (NIC) was a PCI device and it consisted of a CPU (called *LANai*), static RAM (SRAM), and a DMA engine which could move data between the SRAM and the host memory. To send a message, first the LANai processor triggered the DMA to copy the message from the host memory to the NIC memory, then the message was injected to the network. To receive a message, it followed the reverse sequence. Thus, the messages had to be stored in the SRAM in any case. Here, *endpoint* is defined as a object from which a user program can send and receive messages. Endpoints over nodes are logically connected with the others so that an endpoint can send and receive messages to/from any connected endpoints. Here, this set of connected endpoints is called a *channel*. One cannot communicate with the another one belonging to the other channel.

Consider the case, where two independent parallel jobs running on the same cluster use the Myrinet network. A message sent from a job must not be received by the other job. To realize this, each job must have a different channel. So the number of channels means the maximum number of jobs running at the same time. Indeed the size of SRAM in the NIC was very small, only 512 KiB, at the time of SCORE development, and our PM(v2) could provide 13 channels on it. This might sound enough to run 13 jobs at the same time, but for symmetric multiprocessor (SMP) clusters a job could consume the same number of channels with the number of CPU cores (see Sect. 8.2.7). So the number of channels was not enough to run multiple jobs. Instead, we implemented a *virtual channel* where a *physical* channel was multiplexed by saving and restoring the channel status to/from the host memory. This is what we call *network preemption* inspired from the context switching of CPUs for scheduling multiple tasks.

Nowadays a NIC (or often called HCA) can send and receive messages directly from/into the host memory. And a larger number of channels can also be supported (e.g., in InfiniBand). So users do not have to care about the number of the channels and no network preemption is needed any more.

PM's endpoint had a receive message buffer. When a message came in and there was room for the message, the receiver would send back an ACK packet to the sender. When there was no room, it would send back a NACK packet. In normal operation, the sender receiving NACK packets resends the corresponding message. Otherwise, the sender releases the send buffer occupied by the message. When context switching begins, first all message sending is blocked, and then senders would wait for all ACK or NACK packets. When all ACK and NACK packets are received, endpoints are ready for context switching. SCORE-D initiated this network preemption on all nodes of a job and waited for these events in a barrier. In this way, it was guaranteed that the preempted job had no inflight messages.

One could argue that the network preemption is unnecessary if every process to be context-switched has independent end-point(s). PM had several "channels" to be associated with endpoints. When an endpoint is associated with a channel, then it becomes active and it is ready to send and receive. When network preemption takes place, the channels in use are also switched to other endpoints allocated to other

processes. If there was the same number of channels with the number of endpoints and each job exclusively owns its channels, then network preemption would not be needed because no messages would be delivered to the wrong endpoint at the time of context switching. By doing so, however, the program running on the Myrinet NIC should poll all send buffers when the doorbell flag is set to indicate a message is ready to be sent. The processor on the Myrinet NIC was slower than that of the host processor, and this would have added extra latency in message sending. Thus, we eventually decided to have a smaller number of channels than that of endpoints.

Before and after network preemption, user processes of a given job must be stopped and processes of the new scheduled job must be resumed. Since SCore-D was implemented at user level, user processes were stopped by sending the SIGSTOP signal and resumed by SIGCONT.

The network preemption was applied not only for gang scheduling but also for deadlock detection and for checkpoint-restart described in the following paragraphs.

Global State Detection

There was a shared memory region between the SCore-D process and a user process to communicate that a user process may request a service to SCore-D. In this shared memory region, there was a flag to indicate the state of the user process whether it was idle or busy. When a user process had nothing to do but waiting for incoming messages, then the upper level communication layer, i.e., MPI, would set the flag. If the PM(v2) library detected an incoming message the flag was reset.

During job context switching, the saved network context was inspected and if there were no inflight messages found in the network context and all flags of the currently running job indicated idle, the job was considered to be in a deadlock. If there was at least one outstanding request to SCore-D, then the job was considered to be idle waiting for a request and the job would not be scheduled until the request was done (Hori et al. 1997). By detecting user jobs status, idle or deadlocked, SCore-D could avoid wasting valuable computational resources.

Real-Time Load Monitor

In user-level communication, polling is often used to wait for incoming messages. Thus, it is very hard to see from outside of a process if the process is busy for computation or idle just waiting for messages. The idle flag used for detecting the global state of a job could also be used to monitor if a process was busy or idle. SCore-D could display the job's business in real time by sampling the idle flags (shown in Fig. 8.3).

8.2.4 High Availability

The SCore operating system was also designed to be fault resilient. There were two technologies developed; support for checkpoint-restart of user jobs as well as automatic restart of SCore-D global OS.

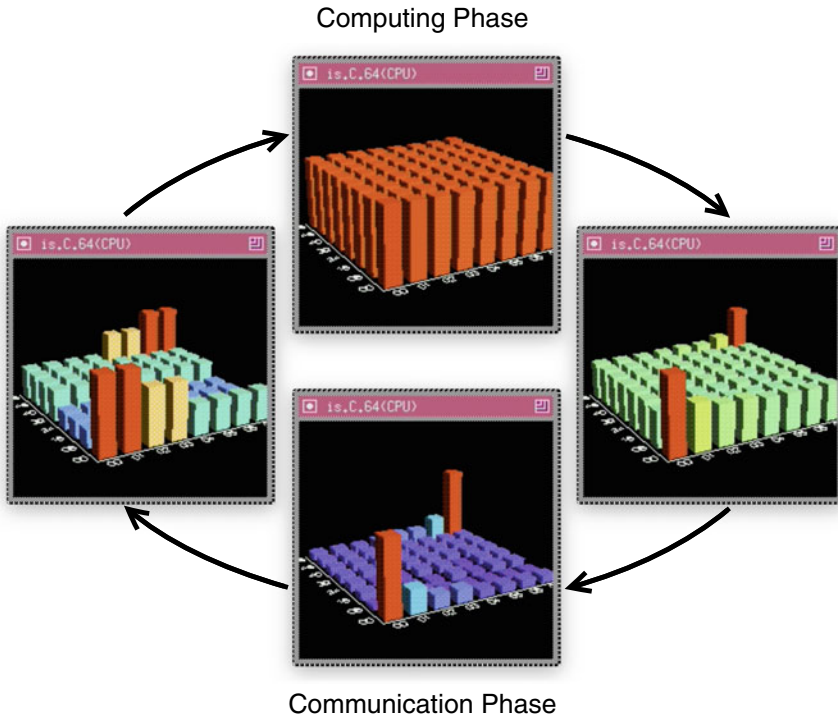


Fig. 8.3 Example of SCore real-time load monitor showing the status of each process. The tall (and red) bar represents the business of a process and the low (and blue) bar represents the idleness of a process. In this example, NPB IS.C.64 was running. Alternations of communication phase and computation phase can be seen

Checkpoint and Parity

A consistent checkpointing mechanism can be implemented by using network pre-emption because network consistency is assured (Nishioka et al. 2000). SCore-D assumed that each node has a local disk, there was no reliable and scalable parallel file system at that time, so process contexts and network contexts were saved to local disks. In case of a node failure, the local disk of the node became inaccessible. To ensure survival from this condition, a unique parity was introduced.

RAID systems having parity are widely used. However, RAID systems require an extra disk for parity. In case of clusters with local disks on each compute node there was no extra disk and RAID-like parity system could not be applied. In the SCore checkpointing system, the idea of parity blocks was very similar to that in RAID systems, however, without the requirement of an extra disk (Kondo et al. 2003).

Figure 8.4 shows how parity was computed and stored into local disks. Every node wrote the process image and network context to its local disk. When a node wrote the first block to the disk, it would send the block to its neighboring node. On the neighbor, the received parity block and the second block to write were XORed

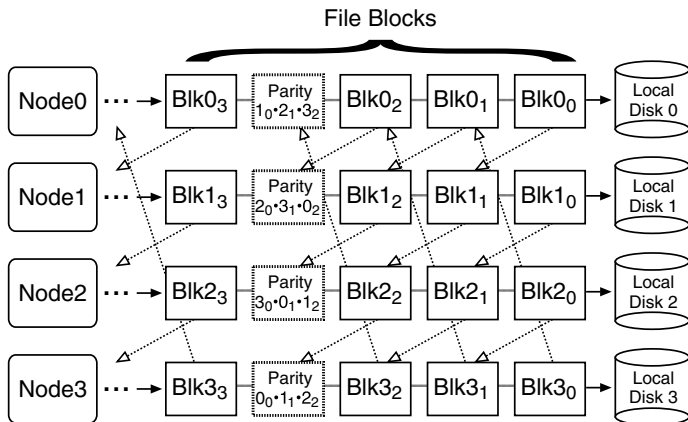


Fig. 8.4 Example of checkpoint parity

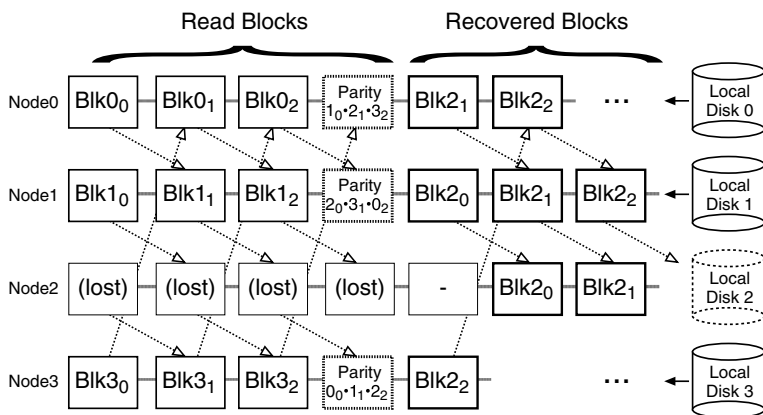


Fig. 8.5 Example of recovery when local Disk2 was lost in Fig. 8.4. The lost file was treated as zero-filled and the procedure for saving checkpoint data with parity was used to reproduce the lost blocks

and this parity block was sent to its neighbor. This procedure was repeated $N - 1$ times, where N is the number of participating nodes, and the received parity block was written to the disk. Similar procedure can also reproduce a lost file (shown in Fig. 8.5).

Automatic Restart of Jobs

Since SCore-D was a global OS, a node failure could result in a deadlock. Fortunately, SCore-D was programmed with MPC++, a multithread C++ extension described in the next subsection and SCore-D had a watch dog timer running as a dedicated thread. A token was passed around among compute nodes periodically. When a time out was detected, all SCore-D processes were shut down. Since this shutdown was done by

the Scout environment (see Sect. 8.2.6), all SCore-D processes running on each node were successfully terminated without resulting in any zombie processes. The jobs running at the time of a shutdown were also killed, however, when the SCore-D was restarted by an operator, those jobs were restarted from the checkpointed images, if any, or restarted from the beginning.

The system operator could specify some nodes in the cluster as spares. If SCore-D was restarted to replace a failed node with a spare node, then processes from the failed node were migrated to the new node set including the spare node.

8.2.5 MPC++

SCore-D was a fairly complex system because each process frequently communicated with others to control and monitor user jobs. From the beginning, we decided to develop a parallel language to write SCore-D. This language was called MPC++, a multithreaded extension to C++.

MPC++ version 1.0 was designed for parallel/distributed programming (Ishikawa, 1996). Instead of setting several language extensions, we have designed a number of low-level parallel description primitives and the MPC++ meta-level architecture to realize an extendable/modifiable programming language system. The parallel description primitives were designed to make the development of the SCore-D operating system as well as complex user applications easy. These primitives were: (a) remote method invocation, (b) synchronization objects, and (c) global pointers.

MPC++ Version 2.0 was designed in two levels, level 0 and level 1. Level 0 specified parallel description primitives realized by the Multi-Thread Template Library (MTTL), C++ templates without any language extensions to define the MPC++ basic parallel execution model. Level 1 specified the MPC++ meta-level architecture and application specific extensions.

There was a user-level multithreaded runtime system implemented for MPC++, named User-Level Thread (ULT), which utilizes the PM communication library. MPC++ and ULT were designed not only for SCore-D but also users to write application programs. Since SCore-D processes ran with user processes, ULT for SCore-D was waiting for incoming messages in blocking operations to avoid consuming CPU time for polling.

8.2.6 Scout Parallel Shell Environment

Scout was a parallel shell execution environment, which was also used to boot up SCore-D processes. Figure 8.6 shows an example of the scout parallel shell execution. The `scout` command with the `-g` option followed by a hostname list separated by the “+” symbol creates a scout session on the listed hosts. On the front-end host where the `scout` command is invoked, a new shell is created. If the `scout` command followed

Fig. 8.6 Example of `scout` parallel shell execution. In this example, the value of the `SHLVL` environment is printed for readers to see the increased `SHLVL` value, indicating a new shell is created in the `scout` session

```
user$ echo $SHLVL
1
user$ scout -g comp0+comp1+ ... +comp6+comp7
SCOUT: session started.
user$ echo $SHLVL
2
user$ scout date
[comp0-3]:
Mon Jan 15 10:10:12 GMT 2018
[comp4]:
Mon Jan 15 10:10:11 GMT 2018
[comp5-7]:
Mon Jan 15 10:10:12 GMT 2018
user$ ^D
SCOUT: Session done.
```

by a normal command is typed in the `scout` session, the command is executed on every host. The unique feature of the `scout` command is that the `STDOUT` messages of each command were merged if the messages of adjacent nodes (see below) were the same. If the created shell terminates, the `scout` session is also terminates.

When a `scout` session is created, `scout` processes are created on each node and they are connected with TCP/IP connections forming a ring topology (shown in Fig. 8.7). The `scout` command invoked by a user becomes a front-end process and forks another shell. Each `scout` process receives the output via `STDOUT` of the executed command and the output is buffered. The `scout` process also receives the output of the previous `scout` process of the ring, and it is compared with the buffered output. If they are the same, then the buffered output is discarded and the received previous output is sent to the next node. If they are not the same, first the local buffered output is sent and then the previous output is sent to the next node. Once this is done, then the previous output is merely forwarded to the next `scout` process until it receives the output of the last node. Merging adjacent messages can reduce the network traffic, and user can easily identify nodes having the same output.

This message merging did not take place when SCore-D was running in a `scout` session. In that case, the output messages of every SCore-D process were merely forwarded to the front-end `scout` process.

8.2.7 From PM to PM(v2)

After the successful development of SCore-D and PM, the SCore development team decided to expand PM to support Ethernet devices and Shmem for intra-node communication.

Ethernet Device

Beowulf clusters used normal Ethernet drivers and the TCP/IP protocol. There is significant overhead in the protocol layers defined by OSI on top of physical protocol

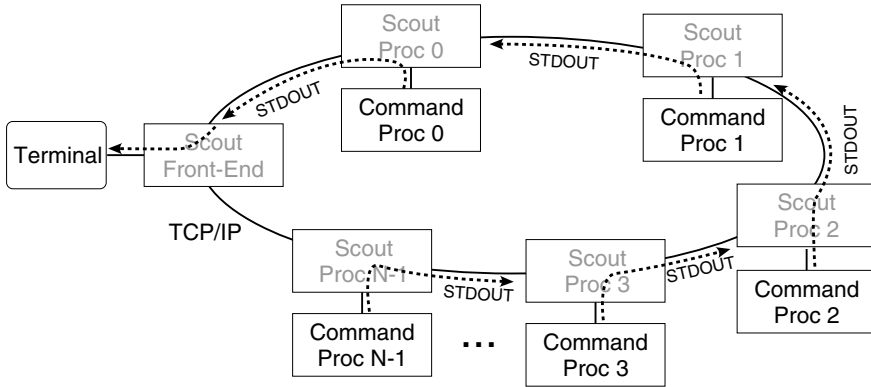


Fig. 8.7 Scout process structure

layer. Our goal was to have a new thin protocol layer optimized for HPC communication. The first implementation was to modify the firmware of the Ethernet NIC. This implementation, called *Giga-E PM*, could roughly double the communication performance, however, the availability of the NIC capable of modifying its firmware was very limited. So we decided to develop a new Linux kernel driver for the thin protocol. This was not a user-level communication because users had to call a system call to send messages and the driver required interrupts raised by Ethernet NIC. However, this approach still doubled the performance on the same Ethernet NIC. This PM device implemented as a kernel driver was named *PM Ethernet* (Sumimoto et al. 1999, 2000a, b).

Intra-Node Communication Device

The first version of the PM(v2) intra-node device used shared memory. However, to send a message, two memory copies must have taken place; one for copying message to the shared memory region and another one to copy the message form the shared memory region to the target buffer. So, we decided to develop a Linux kernel module that allowed a process to expose a memory region to another process resulting in one memory copy to send an intra-node message (Takahashi et al. 1999). This technique is known as kernel-assisted intra-node messaging nowadays. KNEM (Buntinas et al. 2006) and LiMIC (Jin et al. 2005) are implementations of the same idea.

Multi-protocol Support

PM(v2) had a routing table. Although the name might be confusing, this routing table had PM(v2) devices indexed by node number to choose which PM(v2) device to send messages to, not the pass to get the destinations. This routing table is a pseudo PM(v2) device supporting all PM(v2) functions and was named as *PM Composite* (Takahashi et al. 2000). Figure 8.8 shows an example of this routing table.

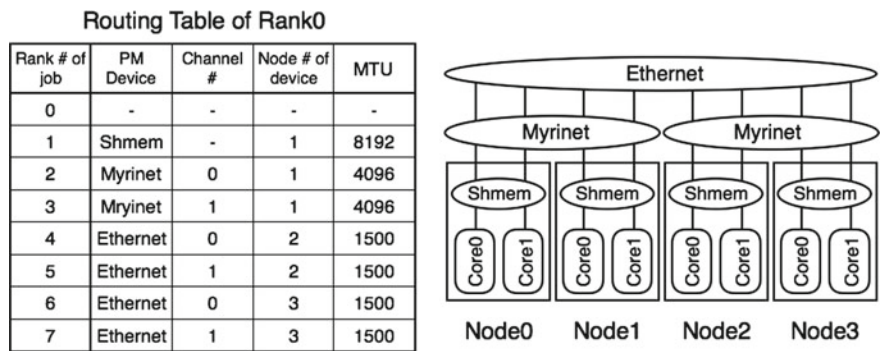


Fig. 8.8 This figure shows a routing table of a small cluster consisting of 4 nodes. Each pair of nodes are connected with Myrinet, and all nodes are connected with Ethernet. Left table shows the example of routing table of the rank 0 process

SMP Cluster Support

To support SMP clusters, which was called *CLUMP* (*CLUster of sMPs*)’ at that time, PM(v2) had a unique feature. Assume that we have two 4-way SMP nodes and running 8 processes in total, one process running on each core. A naive way, which is still used in many cases nowadays, was to have 16 internode connections in whole, with each process having 4 connections to communication with other nodes. Instead of having this all-to-all connection layout, only 4 channels were required in PM(v2). PM(v2) provided connection-less communication and a channel could send messages to any nodes listed in the routing table. Assuming those channels were numbered as *chan0* to *chan3*. *chan0* was used to send message to the first process of each node. *chan1* was used to send messages to the second process, and so on. The channels were shared by sender processes on the same node. So, the maximum number of channels required by a process was limited by the maximum number of CPU cores in a node. Figure 8.8 also shows how the channels were allocated for a CLUMP. This idea was introduced because memory capacity of Myrinet was very limited at that time. In this way, the receiver process was required to poll on only one channel. In contrast, in the naive way, all 16 connections must be polled and this increases communication latency. The number of connections grows rapidly when the number of nodes and the number of processes per node increase.

8.2.8 Heterogeneous Cluster Support

MPC++ was enhanced to support a cluster consisting of different CPU types (Ishikawa et al. 1999). Data marshaling methods were added to the template library. A heterogeneous cluster having 16 x86 CPUs and 16 DEC Alpha CPUs connected with Myrinet and 1G Ethernet was build and SCore was running on it (Fig. 8.9).



Fig. 8.9 Heterogeneous cluster consisting of 16 x86 CPUs and 16 DEC Alpha CPUs. This cluster was demonstrated at the SC'99 exhibition

8.2.9 Upper Software Layer

MPICH-CLUMP

MPICH-CLUMP (Takahashi et al. 1999) was an MPI implementation that had the capabilities of running flat-MPI on SMP clusters (Cappello et al. 2001) as well as zero-copy message transfer (O'Carroll et al. 1998).

SCASH and OMNI OpenMP

The SCore team also developed a distributed shared memory system, named SCASH. SCASH moved memory pages by using PM's RDMA functions (Harada et al. 2000). It was the first time in the world that the Omni OpenMP compiler and SCASH allowed OpenNP programs to run on a distributed memory cluster (Sato et al. 2001).

8.3 Subsequent Developments

On the TOP 500 list released in June 2001, the cluster made by the SCore development team was ranked as 35th. We emphasize, however, that all the faster machines on the list were made by big companies such as IBM, Cray, NEC, Fujitsu, Hitachi, etc.

The RWCP project ended in 2002 and the SCore development team was dismissed, but the SCore software development and maintenance continued by volunteers. The SCore software package and the OMNI OpenMP compiler package were inherited and distributed by the PC Cluster Consortium.¹

After the end of the RWCP project, SCore was installed on some of the major Japanese cluster computers. Streamline Computing, a company based in the UK, also sold SCore clusters in Europe. The DDT parallel debugger developed by Allinea Software, a subsidiary company of Streamline Computing, supported SCore. Most notably, the RIKEN Super Combined Cluster (RSCC) (Kumon et al. 2004), which consisted of a Myrinet cluster and an InfiniBand cluster,² and these two clusters were connected with Gigabit Ethernet. The PM(v2) composite device could utilize Myrinet, InfiniBand and Gigabit Ethernet. The combined cluster was ranked 7th of TOP 500 list in June, 2004.

These look like success stories of SCore, however, the power of research and development of SCore was obviously weakened. The biggest technical issue was the rise of InfiniBand which took over Myrinet eventually. The InfiniBand API known as Verbs does not allow to share an endpoint between processes. Remember that PM(v2)'s endpoint was shared between the SCore-D process and the user processes for gang scheduling. We could not ignore InfiniBand simply because it was dominating HPC. The SCore software structure had to be redesigned to support InfiniBand. SCore Version 7 which is the last major version of SCore series was developed to have InfiniBand support. A new API PM, named PMX (Atsushi Hori, 2009), was also developed. Gang scheduling, the most unique feature of SCore, was sacrificed for having InfiniBand supported.

The author's instinct that time-sharing scheduling was necessary for easy-to-use computers was wrong. In most cases, the SCore-D gang scheduler was not used

¹PC Cluster Consortium: <https://www.pccluster.org>, the SCore package software series can be still downloaded from <https://www.pccluster.org/en/score-download.html>.

²The RSCC cluster was developed by Fujitsu and the InfiniBand made by Fujitsu was used. They also developed their proprietary PM(v2) InfiniBand device, named PM/InfiniBand-FJ (Sumimoto et al. 2004). SCore as an open-source software package did not support InfiniBand at that time.

on big clusters and jobs were scheduled by batch schedulers instead. One reason of using batch scheduling is that users want to solve bigger problems on faster machines. Another reason is that current supercomputers are unable to run jobs in the execution time in which users can patiently watch the execution. The jobs running hours cannot run in minutes on a faster and bigger machine. The benefits of having shorter response time due to gang scheduling were abrogated because of long execution times.

In the 2000s, most computer vendors released HPC cluster products and software packages. In contrast to the situation of SCore, many commercial companies, research firms, and universities were and still are propelling cluster research and development for HPC. SCore's technical advantages were diminished.

8.4 Lessons Learned

One may say the SCore project succeeded in developing new software technologies for HPC cluster computing. Unfortunately, it cannot be said that it succeeded in dominating cluster software. One of the major reasons for this was that the RWCP project ended just before the commercial rise of HPC clusters.

The communication model of InfiniBand is connection oriented. It was obvious for the SCore development team that this would have resulted in having huge number of connections for large clusters. Although later InfiniBand introduced Shared Receive Queues (SRQ) to fix this issue in a limited way. Major MPI implementations also introduced dynamic connections to avoid all connections and use only the connections frequently utilized. This is an example where the technology design in upper stream requires complex treatment in lower stream technologies. Unfortunately, in general, technologies are getting more complex and having deeper layers than ever, which became very hard to optimize as a whole.

The second lesson learned is the importance of international research collaborations. In light of competition among various researchers in the world, only a few software packages with the same objectives could survive at the time of the SCore development. Now the HPC software stack is bigger and deeper than ever and research and development is getting more and more difficult. International research collaboration is getting more active than in the past. We believe this is a good trend.

References

- Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., et al. (1995). Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1), 29–36.
- Buntinas, D., Mercier, G., & Gropp, W. (2006). Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 2006*. (vol. 1, p. 530, pages 10).

- Cappello, F., Richard, O., & Etiemble, D. (2001). Understanding performance of SMP clusters running MPI programs. *Future Generation Computer Systems*, 17(6), 711–720. PaCT. II: HPC applications.
- Harada, H., Ishikawa, Y., Hori, A., Tezuka, H., Sumimoto, S., & Takahashi, T. (2000). Dynamic home node reallocation on software distributed shared memory. In *HPC Asia 2000*.
- Hori, A. (2009). *PMX Specification –DRAFT–*. Allinea Software.
- Hori, A., Tezuka, H., & Ishikawa, Y. (1997). Global state detection using network preemption. In *JSSPP* (pp. 262–276).
- Hori, A., Tezuka, H., & Ishikawa, Y. (1998). Highly efficient gang scheduling implementation. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing 1998* (pp. 1–14). USA: IEEE Computer Society.
- Ishikawa, Y., Hori, A., Tezuka, H., Sumimoto, S., Takahashi, T., & Harada, H. (1999). Parallel C++ programming system on cluster of heterogeneous computers. In *Heterogeneous Computing Workshop* (pp. 73–82).
- Ishikawa, Y. (1996). MPC++ approach to parallel computing environment. *SIGAPP Applied Computing Review*, 4(1), 15–18.
- Jin, H. W., Sur, S., Chai, L., & Panda, D. K. (2005). LiMIC: support for high-performance MPI intra-node communication on Linux cluster. In *2005 International Conference on Parallel Processing (ICPP 2005)* (pp. 184–191).
- Kondo, M., Hayashida, T., Imai, M., Nakamura, H., Nanya, T., & Hori, A. (2003). Evaluation of checkpointing mechanism on score cluster system. *IEICE Transactions on Information and Systems*, 86(12), 2553–2562.
- Kumon, K., Kimura, T., Hotta, K., & Hoshiya, T. (2004). RIKEN Super Combined Cluster (RSCC) system. Technical Report 2, Fujitsu.
- Leiserson, C. E., Abuhamdeh, Z. S., Douglas, D. C., Feynman, C. R., Ganmukhi, M. N., Hill, J. V., et al. (1996). The network architecture of the connection machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2), 145–158.
- Nishioka, T., Hori, A., & Ishikawa, Y. (2000). Consistent checkpointing for high performance clusters. In *CLUSTER* (pp. 367–368).
- O’Carroll, F., Tezuka, H., Hori, A., & Ishikawa, Y. (1998). The design and implementation of zero copy MPI using commodity hardware with a high performance network. In *International Conference on Supercomputing* (pp. 243–250).
- Pakin, S., Karamcheti, V., & Chien, A. A. (1997). Fast messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Transactions on Parallel and Distributed Systems*, 5, 60–73.
- Sato, M., Harada, H., Hasegawa, A., & Ishikawa, Y. (2001). Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. *Scientific Programming*, 9(2,3), 123–130.
- Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawake, U. A., & Packer, C. V. (1995). Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing* (pp. 11–14). CRC Press.
- Sumimoto, S., Naruse, A., Kumon, K., Hosoe, K., & Shimizu, T. (2004). PM/InfiniBand-FJ: A high performance communication facility using InfiniBand for large scale PC clusters. In *Proceedings of Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region* (pp. 104–113).
- Sumimoto, S., Tezuka, H., Hori, A., Harada, H., Takahashi, T., & Ishikawa, Y. (1999). The design and evaluation of high performance communication using a Gigabit Ethernet. In *International Conference on Supercomputing* (pp. 260–267).
- Sumimoto, S., Tezuka, H., Hori, A., Harada, H., Takahashi, T., & Ishikawa, Y. (2000a). GigaE PM: A high performance communication facility using a Gigabit Ethernet. *New Generation Computing*, 18(2), 177–186.

- Sumimoto, S., Tezuka, H., Hori, A., Harada, H., Takahashi, T., & Ishikawa, Y. (2000b). High performance communication using a commodity network for cluster systems. In *HPDC* (pp. 139–146).
- Takahashi, T., O’Carroll, F., Tezuka, H., Hori, A., Sumimoto, S., Harada, H., et al. (1999). Implementation and evaluation of MPI on an SMP cluster. In *IPPS/SPDP Workshops* (pp. 1178–1192).
- Takahashi, T., Sumimoto, S., Hori, A., Harada, H., & Ishikawa, Y. (2000). PM2: A high performance communication middleware for heterogeneous network environments. In *SC*.
- Tezuka, H., Hori, A., & Ishikawa, Y. (1997). PM: a highperformance communication library for multi-user parallel environments. In *Usenix 1997*.
- Tezuka, H., O’Carroll, F., Hori, A., & Ishikawa, Y. (1998). Pin-down Cache: A virtual memory management technique for zero-copy communication. In *Proceedings of the 12th International Parallel Processing Symposium on International Parallel Processing Symposium, IPPS 1998* (p. 308). USA: IEEE Computer Society.
- von Eicken, T., Basu, A., Buch, V., & Vogels, W. (1995). U-Net: A user-level network interface for parallel and distributed computing. *SIGOPS Operating System Review*, 29, 40–53.
- von Eicken, T., Culler, D. E., Goldstein, S. C., & Schauser, K. E. (1992). Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture, ISCA 1992* (pp. 256–266). USA: ACM.